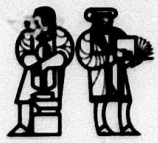
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LEVEL III

Progress Report 17

July 1979 - June 1980

Prepared for the

Defense Advanced Research Projects Agency

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: LCS Progress Report 17

2. GOVT ACCESSION NO.: AD-A093 384

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Laboratory for Computer Science Progress Report 17, July 1979 - June 1980

5. TYPE OF REPORT & PERIOD COVERED: DARPA-DOD Progress Report 7/79-6/80

6. PERFORMING ORG. REPORT NUMBER: LCS/PR-17

7. AUTHOR(s): Laboratory for Computer Science Participants; Michael L. Dertouzos, Director

8. CONTRACT OR GRANT NUMBER(s): N00014-75-C0661

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Laboratory for Computer Science / Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Ma. 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: DARPA Order-2095

11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, Va. 22209

12. REPORT DATE: December 11, 1980

13. NUMBER OF PAGES: 178

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va. 22217

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CLU
computation structures
computer networks
computer systems
database systems
distributed systems
dynamic modeling
information systems
knowledge based systems
local networks
multi-access computers
natural language
network protocols
on-line computers
personal computers
programming languages
real-time computers

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

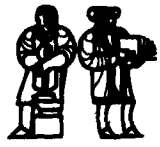
This report summarizes the research performed by the MIT Laboratory for Computer Science from July 1, 1979 to June 30, 1980.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

409648

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Progress Report 17

July 1979 - June 1980

Prepared for the

Defense Advanced Research Projects Agency

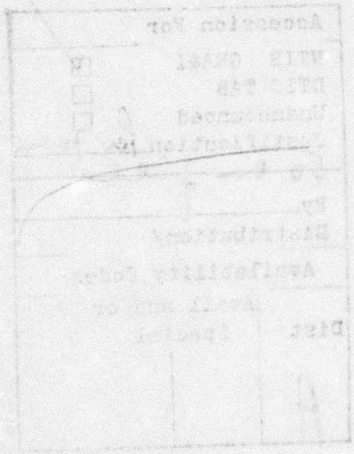
Effective date of contract:	**1 January 1975**
Contract expiration date:	**31 December 1980**
Principal Investigator and Director:	**Michael L. Dertouzos (617)253-2145**

This research is supported by the Defense Advanced Research Projects Agency under Contract No. N00014-75-C-0661, DARPA Order No. 2095 and No. 3795.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

TABLE OF CONTENTS

TABLE OF CONTENTS



ADMINISTRATION

Academic Staff

M. L. Dertouzos	Director
M. Hammer	Associate Director
A. R. Meyer	Associate Director

Administrative Staff

J. Badal	Information Specialist
M. E. Baker	Administrative Assistant
P. G. Heinmiller	Librarian
E. I. Kampits	Administrative Officer
T. E. Lightburn	Fiscal Officer

Support Staff

G. W. Brown	D. Kontrimus
L. S. Cavallaro	E. M. Profirio
C. Cornish	P. Vancini
M. J. Cummings	

Work reported herein was carried out within the Laboratory for Computer Science, an MIT interdepartmental laboratory. During 1979-1980 the principal financial support (55%) of the Laboratory has come from the Defense Advanced Research Projects Agency (DARPA), under Office of Naval Research Contract N00014-75-C-0661. DARPA has been instrumental in supporting most of our research during the last 17 years and is gratefully acknowledged here.

Assembly and final editing of this report was done by J. Badal, with assistance from V. E. Golden and J. Schoof.

Introduction

The Laboratory for Computer Science (LCS) is an MIT interdepartmental laboratory whose principal goal is research in computer science and engineering.

Founded with DARPA funding in 1963 as Project MAC (for Multiple Access Computer and Machine Aided Cognition), the Laboratory developed the Compatible Time-Sharing System (CTSS), one of the first time-shared systems in the world, and Multics--an improved time-shared system that introduced several new concepts. These two major developments stimulated research activities in the application of on-line computing to such diverse fields as engineering, architecture, mathematics, biology, medicine, library science, and management. Since that time, the Laboratory's objectives have expanded, leading to research across a broad front of activities that now span four principal areas.

The first such area, entitled Knowledge Based Programs, involves making programs more intelligent by capturing, representing, and using knowledge which is specific to the problem domain. Examples are the use of expert medical knowledge for assistance in diagnosis and for drug administration carried out by the Clinical Decision Making research group; the use of mathematical knowledge by the Mathlab research group for an automated "mathematical assistant"; and the use of knowledge in programs that comprehend typed natural language (English) queries. Of the above examples, the latter is funded by DARPA.

Research in the second area, entitled Machines, Languages and Systems, strives to effect sizable improvements in the ease of utilization and cost effectiveness of computing systems. For example, the Programming Methodology research group strives to achieve this broad goal through research in the semantics of geographically distributed systems. Toward the same goal, the Real Time Systems group is exploring distributed operating systems and the architecture of single-user powerful computers that are interconnected by communication networks. Other research examples in this area include the study of data bases, and the architecture of very fast multiprocessor machines by the Computation Structures research group, to link large numbers of otherwise autonomous computers. This research is supported entirely by DARPA.

The Laboratory's third principal area of research, Theory, involves exploration and development of theoretical foundations in computer science. For example, the Theory of Computation research group strives to understand ultimate limits in space and time associated with various classes of algorithms, the semantics of programming languages from both analytical and synthetic viewpoints, the logic of programs and the links between mathematics and the privacy/authentication of

computer messages. The Theory research is funded primarily by the National Science Foundation.

The fourth area of Laboratory research is entitled Computers and People and entails societal as well as technical aspects of the interrelationships between people and machines. Examples of research in this area include office automation research carried out by the similarly named research group; the use of interconnected computers for strategic planning; as well as the sociological impact of computers on individuals and the ethical problems of distributed responsibility posed by multiprogrammer systems. Of these activities the strategic planning research is funded by DARPA.

During the past year, the Laboratory consisted of 277 members--39 faculty, 8 visiting faculty, 15 visitors, 90 professional and support staff, 85 graduate and 40 undergraduate students--organized into 15 research groups. The academic affiliation of most of the faculty and students is with the Department of Electrical Engineering and Computer Science. Other academic departments represented in the Laboratory membership are Mathematics, Architecture, Division for Study and Research in Education, Humanities, and the Sloan School of Management. Laboratory research during 1979-80 was funded by 13 governmental and industrial organizations, of which the Defense Advanced Research Projects Agency provided about sixty percent of the total research funds.

The 1979-80 year was very active. Technical results were disseminated through the publications of the Laboratory members and will not be discussed here. Highlights of the year included the following:

During the reporting period, Professor Michael Hammer was appointed Associate Director of the Laboratory, joining Professor Albert Meyer who is also Associate Director of LCS--the need for two associate directors is the result of the Laboratory's rapid growth and increasing involvement with industry. Also during the reporting period Senior Research Scientist Albert Vezza was appointed coordinator of all LCS computational resources--an action made necessary by the increasing number and diversity of our computers and associated equipment.

During 1979-80 LCS has started developing certain VLSI tools in preparation for substantial VLSI research acitivity in 1981. In this area our mission will be the development of design tools for VLSI and pursuit of special-structure architectures for VLSI circuitry. We expect this to be a major research theme for our Laboratory for several years to come.

Our major Laboratory focus on geographically distributed systems has continued

to occupy the attention of more than half of our personnel. First we have completed our design of a powerful personal computer that can employ different microprocessors as the technology of the latter progresses. This design was successfully transferred to Zenith Data Systems who delivered ten engineering prototypes to us. In the coming years we expect to acquire a total of 150 such "advanced nodes" which we will use as direct research vehicles in some seven laboratory research groups.

Our research in distributed computing can be viewed as a search for equilibrium between the opposing forces toward centralization and decentralization - centralization since it maximizes order by vesting authority in one locus, and decentralization because of people's inherent need to control and use their own resources. We believe that increasing decentralization will have a significant effect on the field of computing in that: (1) it will make possible larger numbers of intercommunicating computational resources, and (2) it will permit acceptable operation of the aggregate system in spite of failures of local nodes.

During the reporting period, a new group called Systematic Programming was formed by Professor Guttag. His research strives to establish a language in which program specifications can be effectively expressed, thereby improving the efficiency of large program development. We expect this group to grow during the coming year and will propose that it be funded by DARPA.

The Laboratory's Distinguished Lecturer Series, initiated in 1976, has proved very successful in attracting members of the MIT community. The 1979-80 lecturers under this series were: John McCarthy (Director, Artificial Intelligence Laboratory, Stanford University), Lawrence G. Roberts (Chairman, Telenet Communications Corporation), Kenneth E. Iverson (IBM Fellow, IBM Thomas J. Watson Research Center), Jacob Schwartz (Professor of Mathematics and Computer Science, Courant Institute of Mathematical Sciences, New York University), Brian Randell (Professor of Computing Science, Computer Laboratory, The University of Newcastle upon Tyne), and Dana Scott (Professor of Mathematical Logic, University of Oxford).

During 1979-80, research in previously established areas yielded several new results which were published through Laboratory technical reports (TR219-TR239) and technical memoranda (TM138-TM167), as well as through articles in the technical literature.

Michael L. Dertouzos
Director

COMPUTER SYSTEMS RESEARCH

Academic Staff

J.H. Saltzer, Group Leader
D.D. Clark
F.J. Corbato
I. Greif
D.P. Reed
L. Svobodova

Research Staff

J.N. Chiappa
E.A. Martin

Graduate Students

W. Ames
G. Arens
R. Baldwin
E. Ciccarelli
G. Cooper
W. Gramlich
S. Kent
V. Ketelboeter
L. Lopez
A. Luniewski
A. Mendelsohn
B. Myers
R. Schiffenbauer
C. Seaquist
V. Singh
K. Sollins
P.T. Tung

Undergraduate Students

L. Allen
G. Balikrishnan
A. Chiang
D. Daniels
K. Khalsa
J. Lucassen
C. Ludwig
J. Marrgraff
A. Mondori
P. O'Donnell
H. Peterson
M. Plotnick
W. Rubin
G. Simpson
M. Smith
G. Stathis
D. Theriault
J. Thomas
S. Toner
R. Vieraitis
E. Wylen

Support Staff

R. Bisbee
D. Fagin
J. Jones
E. Poithier
M. Webber

Visitors

N. Natarajan
M. Sinha

COMPUTER SYSTEMS RESEARCH

1. OVERVIEW

The Computer Systems Research Group has been working on several pragmatic aspects of what is loosely called "distributed systems." At the bottom of the implementation structure, the group has been exploring the ring approach to local networks. At the next level up, network interconnection to create a campus-wide area network was a major research and service activity. At a higher protocol level yet, design and implementation of the SWALLOW distributed data storage system, a storage service with unique semantics designed for transaction reliability was begun. Finally, exploration of distributed systems applications involved implementing a distributed calendar system. These projects, among others, are described in detail in the following subsections.

2. LOCAL NETWORK TECHNOLOGY RESEARCH

Local network technology in the United States is dominated by variations on the passive broadcast cable pioneered by the Xerox Palo Alto Research Center Ethernet. An alternative technology, the ring of active repeaters, has received less attention, even though it offers a number of attractive properties, including simpler analog engineering, ability to cover a larger geographic area, ability to use fiber optics, and ability to scale up to very high speeds. The laboratory has a modest project underway to explore this alternative in more depth, and to learn more about the properties of the ring network in the field.

2.1. Prototype Ring Experience

A prototype ring network, running at a data transmission rate of 1 Mbit/sec has now been in operation in the laboratory for 18 months, currently connecting eight PDP-11, LSI-11, and VAX computers including a bridge to the other local networks. This basic ring structure has proven to be quite effective in day-to-day use, although the need for automatic reconfiguration when nodes are taken down has been clearly demonstrated. (Reconfiguration in the prototype ring is done manually from a central location.) A second copy of the prototype ring was installed at University of California, Los Angeles (UCLA) in November 1979, and has been operating there quite effectively, also. (So far, neither of these installations has stressed the ring capabilities enough to provide convincing demonstration of the concept, though.)

As part of the evaluation of the prototype ring, an undergraduate thesis was

completed [Vieraitis] that involved implementing a network performance monitor and collecting an initial set of data. Statistics and operational characteristics quite similar to those reported for the Xerox PARC Ethernet were observed.

2.2. Version 2 Ring

In conjunction with a subcontractor, Proteon Associates, Inc., the prototype ring network was reengineered around a simpler design and for a transmission rate of 8 Mbit/sec to produce what is called the Version 2 ring. The hardware for the Version 2 ring is designed to fit into a general system for a local area network that can cover a site such as the MIT campus.

The key to this system is definition of a high speed byte-parallel local network interface that permits, on one side, implementation of any of several local network technologies, and on the other, implementation of buffered channel or bus attachments for any of several computers.

Thus the Version 2 ring controller comprises a modem, clock circuits, token and ring format management, all on a 5 by 8 inch card containing about 30 TTL integrated circuit packages; it attaches by connector to the standard interface. One lesson learned from checkout of the prototype ring was the value of built-in checkout features; the Version 2 ring controller includes a 10-bit shift register that can be connected from the transmitter to the receiver in place of the rest of the ring, allowing local checkout of almost all features of the controller.

Clock coordination is probably the single hardest problem to accomplish in a ring when the goal is to avoid dependence on a central or special station. Agreement on the exact frequency of data transmission must somehow be reached collectively. (In both the Cambridge University and Toshiba high speed ring networks a central station sets the clock rate.) Two schemes have been investigated, with special interest in their stability at high data rates and with large numbers (say 200) of nodes. The initial implementation uses a frequency-adjusting phase-locked loop in each node, comparing the observed received data rate with the local clock and fine-tuning the local (crystal) oscillator to match. A string of repeaters thus would all synchronize their clocks to the frequency of the first node in the string; a closed ring will home in to a communally-agreed-upon average frequency, with the possibility of oscillation around that frequency that can apparently be damped by appropriate choice of filter values in the individual phase-locked loops. Two simple, first-order mathematical analyses predict that stability is easily accomplished; field experience will be required to learn how closely these first-order models reflect the actual operating environment.

A second clock coordination scheme, extrapolated from the scheme used

successfully in the prototype ring, is also being investigated. In this second approach, the local clock of each node runs at some modest multiple, say 6X, of the nominal data transmission rate, but its frequency is fixed. Received data is examined and its clock rate and phase extracted and compared with that of the transmitter side of the node. If the transmitter phase drifts more than, say, 1/6 of a bit time ahead or behind the received phase, the transmitter sends one bit that is either 1/6 of a bit time shorter or longer than usual, so as to catch up. This approach has the virtue that it is largely digital in nature, can correct much larger frequency errors, and does not require continuous transmission. However, for stability it requires that between messages there should be gaps with no transmitted data, which in turn requires the receiver of a node be able to decode incoming data starting with the first transition of a sequence of bits; at high frequencies and in the presence of noise, this rapid startup is relatively hard to accomplish.

As mentioned above, the frequency-adjusting, closed loop design is being used in the initial implementation. As a second, parallel effort, the phase-adjusting scheme is being tested for its potential applicability.

The frequency-adjusting modem, data transmission over 800 feet of twinax cable, and the ring controller have all been demonstrated individually and in a 2-node ring and their successful integration is expected to be imminent.

2.3. Other Local Area Network Components

As mentioned, the Version 2 ring is designed as part of a general, modular system for a local area network. Several other components of this system have been imagined, designed, or implemented. On the host computer side of the byte-parallel net interface, a full-duplex, buffered, direct memory access module for the PDP-11 UNIBUS was specified, designed, implemented, and checked out. Two copies of this 100-chip card have now been built. Similarly, a buffer module for the S-100 bus has been specified, and design begun, and buffer module implementations are planned for the Nu-bus and the Q-bus; recently a proposal to implement a buffer module for an IBM 370 channel was discussed.

On the other side of the byte-parallel net interface, design has just begun on a "long-distance bridge" module, which would allow interconnection of local nets in different buildings. The initial version of this bridge will probably use the same basic ring control strategy as the Version 2 ring with minor specialization to the case of two nodes and long cables; options such as fiber optic technology are also being examined.

Other possible network technologies that could easily be attached as part of this same system include a packet radio network for communication with computers

located in private homes, an Ethernet based on the recently announced standard agreed upon by Xerox, Digital Equipment Corporation, and Intel Corporation, and an X.25 interface to TELENET or TYMNET. Each of these, in turn, could be directly attached to any host for which a host-specific buffer module had been implemented.

3. PROTOCOL DESIGN AND NETWORK INTERCONNECTION

Currently, there are five different network technologies deployed or under development in the laboratory, and four different protocol families in use, with more on the horizon. This excessive wealth of material raises problems of substantial theoretical interest, which must be immediately solved if we are to provide any sort of stable service to the laboratory community.

As a practical matter, the proliferation of network hardware is less disruptive than the proliferation of protocols. Our assumption has been that experimentation with network hardware technology is healthy and appropriate, but that protocol standardization is important if the various machines in our laboratory are to be able to communicate. Thus, we have been attempting to standardize the laboratory on the protocol family developed by the ARPA internet working group, variously called internet or transmission control protocol (TCP). Implementations of these protocols have either been implemented or imported for the Multics system, UNIX, Tops 20, and the Alto. The Alto implementation is coded but not debugged, the other implementations are operational, at least for friendly users. The function of this protocol is to permit traditional services such as remote login, file transfer, and mail to operate in the local environment. Our group has also specified an extremely simple file transfer protocol, as an interim measure until the TCP based file transfer protocol is generally available. This protocol, called trivial file transfer protocol (TFTP), has been implemented for Multics, UNIX, Tops 20, the Alto, and as a stand alone program suitable for downloading a PDP-11. This protocol will permit the transfer of files and mail between the above mentioned machines, and is also the basis for the UNIX access to the Dover.

A subnet gateway has been implemented and placed in operation between the local Ethernet, the Version 1 ringnet, and the Xerox Ethernet. This gateway is in regular use, providing communication between the 11/70 and the Dover spooler, and between the VAX and machines on the Chaos net. Measurements over a recent 24-hour period indicated a total traffic through the gateway of approximately 14,000 packets.

As part of this project, it has been necessary to develop a number of specialized tools, including a fairly sophisticated workbench on the UNIX system for the creation of programs for stand alone PDP-11s. Software now exists which allows us to

combine programs written in assembly language, C, and BCPL, all languages which have been used to write programs which we needed to import.

Several slightly longer range projects have also been completed:

- Jerry Saltzer has written two memos outlining a possible approach to networking an environment such as the entire MIT campus.

- Hal Peterson did a study of the congestion control mechanism currently implemented in TCP, a study which indicated certain potential problems with this area of the protocol.

- Will Ames built a modular simulator that can be used to study performance of different network configurations under different high-level protocols. Ames performed a number of simulation experiments for a ring network with 60 and 120 nodes running TFTP. He studied the effects of the ring speed and packet size on the response time and throughput of the whole system.

- Kirpal Khalsa completed a preliminary study of specialized flow control algorithms for file transfer protocols.

Several bottlenecks remain, most notably getting XX on the local net and connecting the local net to the ARPANET. DARPA has agreed to deliver an additional IMP to solve this latter problem, and we are importing a Port Expander as a short-range solution.

4. XEROX UNIVERSITY GRANT

During this reporting year the Xerox Corporation, stimulated by proposals from the Xerox Palo Alto Research Center, initiated a university grant program that supplied MIT, Stanford, and Carnegie-Mellon University each with 18 "Alto" personal computers, a "Dover" laser-driven xerographic output printer, an Alto-based file storage system, an Ethernet local network, and a large quantity of supporting interactive software. Installation of most of the equipment was completed by February 1980, and bridges between the Ethernet and the other local networks were rapidly developed to allow access to the Dover printer from other computer systems.

The initial use of this equipment has been largely explorational, based on the supplied software, which among other things provides advanced word processing and illustration facilities. The impact of just these facilities, together with the Dover, was clearly noticeable during the spring thesis season. Quite a number of recent

theses, technical reports, and papers have been prepared with this equipment, and nearly all text processing output of the Laboratory for Computer Science and Artificial Intelligence Laboratory now is printed on the Dover, which is consuming 150,000 sheets of paper per month. A noticeable increase in the number of illustrations, drawings, and graphs in reports and memos seems to have accompanied use of the Alto report preparation software.

A substantial library of computer games has migrated from other Alto sites. As one might expect, these games are taking a certain toll in graduate student time and attention, although they turn out to be a less serious hazard to academic and research interests than one might expect. Instead, since many of these games demand rapid interaction, they also reveal limitations and requirements for highly interactive software, and on the whole are probably a cultural benefit of the grant. In a similar way, the use of the other software systems is providing both a feel for the depth of engineering required to create a good human interface and an inspiration for some enterprising activists to do better in local implementations of some of the same ideas.

Primarily because of the current availability of LISP machines and expected imminent availability of Nu computers, enthusiasm for starting major new programming projects in the Alto environment has been quite low. The programming projects that have started are limited in scope or special in nature:

1) David Reed and Liba Svobodova are supervising design and implementation of the Swallow distributed data storage service on an Alto that can be equipped with several hundred megabytes of disk storage. The research goal of this project is described elsewhere in this annual report. The primary reason for use of the Alto environment is immediate availability of both disk hardware and the Mesa programming system, together with an estimate that the initial implementation will fit easily in the Alto memory space.

2) David Clark has implemented Internet and associated file transfer protocols for the Alto, to allow communication between the Xerox grant equipment and the other computers in the laboratory and the ARPANET community. In conjunction with these protocols, he has deployed a Dover printing service.

3) Robert Schiffenbauer is developing a Mesa-based subsystem for debugging distributed applications.

4) John Guttag is supervising the programming (in Mesa) from formal specifications of a Bravo-like display interface. The purpose is to

understand better the implications for programming and system design of working top-down with formal specifications.

During the coming year a few more research projects are expected to begin using this equipment: a programming specification verification system, some VLSI circuit design work using the ICARUS system, and a bootstrapped CLU compiler have all been discussed.

5. THE SWALLOW DISTRIBUTED DATA STORAGE SYSTEM

5.1. Overview

The Swallow project was begun last summer. Its purpose is to design and implement a coherent organization for long-term storage in a network of computers. We assume that these computers are managed in a decentralized way, preserving for each computer in the network a high degree of autonomy. In particular, we would like to obviate any need for a central authority (human or computer) that has complete control of the activities and data in the network. Thus, unlike traditional computer operating systems in which the supervisor manages all computational and memory resources, our distributed environment is much more like a loose coalition of computers that frequently need to cooperate and to share information, but which computers control completely how they cooperate.

In this context, Swallow can be viewed as a set of standard protocols that cooperating computers may use to manage their data. If Swallow is to be successful in this environment, it must both provide benefits when used and not compromise the autonomy (of course, it must compromise autonomy to the extent of requiring certain standard interfaces).

The benefits Swallow provides include the following:

- **Uniform interface** - the read and write operations provided to users of Swallow make the location of data stored in the system transparent. The owners of data are allowed to control the location of data, however.

- **Reliability** - Swallow provides storage for data objects that is extremely stable. In addition, only those nodes that hold data needed by a computation need be available to run that computation, so availability is enhanced.

- **Atomic Actions** - Swallow provides synchronization and recovery mechanisms so that any arbitrary set of accesses may be combined into

an atomic action, using the model developed by Reed [1,2]. Network failures and node crashes do not compromise proper synchronization and recovery of these atomic actions.

- **Protection** - a standard mechanism for encryption-based protection of data stored in the system will be provided. This mechanism is decentralized, so that there is no critical central authority that can compromise the security of every user of Swallow.

- **Support for "small" objects** - novel organizations of storage are needed to support the object model proposed by Reed; at the same time, such storage organizations can be designed to support small objects effectively. The user of Swallow sees an environment consisting of a large number of objects whose average size is relatively small.

These properties are synergistic. For example, in a traditional file system, it is usually not possible to perform atomic actions that involve multiple files. Consequently, objects accessed within the same atomic actions must be stored in the same file. This is one reason that files are large. In the Swallow system, since atomic actions may access multiple objects, it is quite reasonable to store "files" as structures consisting of many individual objects.

5.2. Overall Structure of the Swallow System

Figure 1 illustrates the overall structure of the Swallow system. Each client computer that uses Swallow accesses storage via a module called the broker, which is implemented on each client. The data owned by that computer is stored either on local secondary storage or on a shared server called a **repository**.

The broker has two functions--it controls the location of, and mediates all accesses to, data owned by its client.

The repository provides large quantities of stable storage. To simplify the job of the repository, a repository is not responsible for protecting the data stored there from unauthorized release.

Both the brokers and the repositories support the protocols needed to provide atomic actions, since both types of modules contain objects that may be used by atomic actions.

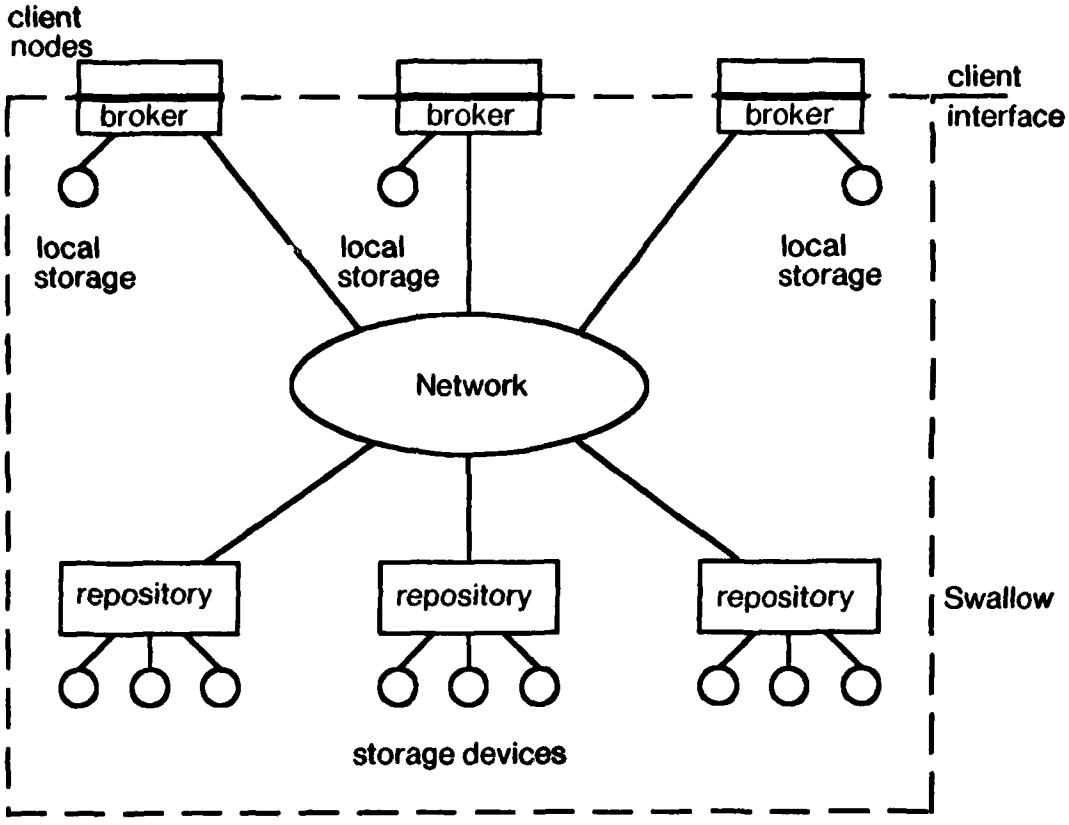


Figure 1-1: Swallow System Structure

5.3. Management of Objects in the SWALLOW Repository

The repositories provide stable, reliable, long-term storage for the client nodes. They must handle efficiently both very small and very large objects and incorporate mechanisms for updating a group of objects at one or more physical nodes in a single atomic action. The repositories support, with some minor modifications, the object model developed by Reed. This model provides the basis for synchronization and recovery in the implementation of atomic actions.

An object in a repository is represented as a history of all the states it has assumed since its creation. Each distinguishable state is represented by a special immutable entity called a **version**. In the actual implementation, an object consists of a mutable object header and an append-only linked list of versions. Associated with each atomic action is a **commit record** that records the status of that action (unknown, committed, or aborted), and groups together all objects modified by that action. Inside the repository, commit records are represented by objects.

The core of the repository is a write-once storage model called Version Storage (VS), which gives an illusion of an append-only infinite tape, but supports random read accesses. VS stores information as stable immutable entities that are called **VS images**. VS is the only stable storage in the repository. It contains the histories of all objects in the repository. In addition, all the information needed for a crash recovery must be stored in VS, as immutable VS images. In a sense, VS is similar to the transaction log of database management systems [1]. However, there is an important difference: VS is used not just for recovery, but it is where the actual data are.

VS provides a linear paged address space with a straightforward mapping from the VS address into a location on the physical device. VS is duplicated for stability, but since no update in place is possible, the two required writes can be concurrent. For easier management of VS (mainly for faster VS address resolution and object location), VS is allocated in fixed-sized pages, which are the units of **atomic write** into VS. Large objects are stored as **structured versions** that are constructed successively as fragments of an object and delivered by the communication subsystem. The repository never has to assemble the entire object before it can start writing it into VS: received fragments, regardless of their size, are processed immediately and stored as VS images.

Since VS may grow very large, it is impossible to maintain the entire VS online. Only the upper 2^n words of VS are kept in the Online Version Storage (OVS). OVS would thus contain the current versions and tokens of the recently updated objects. To make sure that the current versions of most objects are found in OVS, it is necessary to copy occassionally the images of current versions and tokens to the high end of VS. Two different policies for retaining such images in OVS were investigated: one policy is to keep the current versions of all objects in OVS; the other is to keep in OVS only the current versions of those objects that have been used in the recent past. Implementation of the first policy is rather complex and costly. Thus, given that it is likely that many objects will be dormant for long periods of time, the second policy is preferred. When the current version of an object is read, the OVS manager will make a copy of it (create a new VS image) if the VS address of the read image falls past a copy mark. The position of the copy mark changes as new images are created in VS; the actual implementation further depends on the type of storage devices used. This scheme preserves locality of reference, and automatically brings back online the current versions of the objects that have not been used for a long time.

OVS can be implemented with a reusable device, or with write-once devices. The latter form simplifies the transfer of VS images from online to offline storage. The delays due to manual device replacement can be eliminated through a circular assignment of device drives to different functions in the implementation of OVS.

A critical concern addressed throughout the design of the repository is recovery from system crashes and storage device failures. The problem of storage device failures is solved primarily by duplicating all information stored in VS and by using implicit bypasses for bad areas on the physical storage devices so as not to disrupt the sequential nature of the VS address space.

The crash recovery of the repositories is based entirely on the information contained in VS. Current contents of object headers, although the object headers are the key elements in all operations on objects, are, for performance reasons, treated as hints that are fully reconstructable from the information found in VS. The commit records are implemented as objects, and thus are reconstructable by the same process. Finally, the object directory in the repository is an object itself and hence reconstructable from the information in VS.

The recovery of the entire repository requires a complete scan of VS, starting from the last page of VS. Individual VS images are isolated and examined. Once the current version of an object has been found, the object header can be easily reconstructed. Unfortunately, when this sequential search of VS finds the first image for a particular object, it is not necessarily an image of the current version. This may happen because of the copying in OVS, but also because of the way small VS images are packed into pages prior to being written into VS. Consequently, the algorithm for reconstructing the object headers is quite complicated.

The recovery can be distributed over time, such that the recovery process is invoked for one object at a time, as individual objects are accessed. A special checkpoint (a VS image of the object header) is created when an object is recovered; at any point, the VS search needed to recover a single object is limited to a single **recovery epoch.**

A technical report on "Management of Object Histories in the SWALLOW Repository" by Liba Svobodova is in preparation.

5.4. Implementation

We are implementing Swallow to show that the concepts involved (uniform interface, atomic actions, ...) can be used in a practical system. Our primary concerns are efficiency and usability. Since the organization of Swallow is radically different from traditional storage systems, the only way to understand how well it will perform in practice is to build it, and then use it in constructing some applications.

Our goal is to implement a prototype system with most of the features of Swallow on a set of Altos, with at least one repository node, and several brokers/client nodes. Altos were chosen because of the existence of both solid hardware and well-

developed support software (e.g., Mesa). As the Nu machines become available, we will migrate the system onto the Nu's, first constructing a broker for the ECLU environment on the Nu's, and eventually constructing Nu-based repositories.

Our efforts for the past year have been aimed at creating an implementable design. The first design phase is nearly complete--we have the following pieces to build.

- **Message protocol** supports datagram service for messages of arbitrary length.

- **Object access protocol** coordinates interactions between brokers and repositories.

- **Version storage management** manages secondary storage (magnetic or optical disk and tape) used for holding the versions of objects.

- **Object history manager** maintains the history of versions of objects. Implements stable storage and recovery mechanisms for crashes.

- **Repository control** supervises execution of transactions on the repository.

- **Commit record manager** implements the two-phase commit protocol among repository and client nodes.

- **Broker control** keeps track of objects owned by the broker.

The interfaces and algorithms for these modules have been developed over the past year. During the coming summer we plan to implement these.

5.5. Major achievements this year

G. Arens has defined the object access protocol used between brokers and repositories. This protocol is "connectionless," that is, the only state information maintained at each node is the values of objects. Since there is no connection state, there is no delay in initiating communications.

L. Svobodova has designed the part of the repository that manages objects and commit records. This design was discussed in the section entitled "Management of Objects on the SWALLOW Repository."

D. Reed has developed an approach to protecting objects using encryption. Objects stored on the repositories are encrypted, with keys known to the owning brokers only. Thus there is no need for implementing a common access policy on the repositories. This simplifies the repositories, and allows clients flexibility to implement arbitrary access control policies.

6. THE AUTHENTICATION SERVER

The authentication server project has two goals. The first is to build a key distribution center that can be used to support other distributed system components that are to be built here. In particular, the Swallow system, described above, will store data in an encrypted form and will therefore require such a server. Also, any secure conversation between processes in the system might require similar services. The main function of the authentication server will be to provide for key distribution.

A second purpose of the project has been to provide a source of experience with programming for a distributed system. The currently available "extended" CLU has served as the language for several experimental implementations. By reviewing our programming experiences regularly, we are developing some insight into how such a language can support the implementation of programs for a distributed environment.

We began meeting in September 1979, before any other projects had developed detailed specifications of their authentication server requirements. We spent about two months reading selections from the literature on protection and encryption, as well as learning "extended" CLU. At that time we decided to implement the protocols for establishing a secure conversation as presented in Needham and Schroeder [3]. There are two versions of the protocols, the first for use with conventional encryption and the second based on public key encryption. These protocols should have some relationship to protocols required by local users, but are not particularly tailored to the needs of other projects in the laboratory. Thus the main results of this exercise have been initiation into the extended CLU programming environment, production of two simple servers that can serve as foundations upon which to build, and identification of a variety of problems not addressed in the Needham and Schroeder paper.

The next phase of the work was to redesign the programs so that communication could proceed in terms of internal datagrams. The group defined a datagram standard for use with the Needham-Schroeder protocols and have begun a new implementation.

There are three kinds of future work that we are considering. First, there are still parts of the current implementation of the Needham and Schroeder work that are

incomplete. The encryption procedures do not implement secure encryption algorithms. Also, Needham and Schroeder suggest some modifications to their protocols that would facilitate caching of keys for reuse in future conversations. The current implementations require that the authentication server be involved each time a new conversation is started.

Second, there are issues that were outside the scope of the Needham and Schroeder paper that we can tackle. These include protocols for proceeding with a conversation once a key has been agreed upon and protocols for revoking a key once it has been compromised.

Third, we are interested in providing services that will be of use to people building other programs. For example, if the data storage server provides storage for large numbers of small objects, each under a separate key, then adequate performance may depend on its ability to get a large number of keys from the authentication server in response to a single request.

7. APPLICATIONS FOR DISTRIBUTED SYSTEMS

7.1. The Application

In the area of applications we have continued to focus on distributed calendar systems. There are two kinds of calendars that we have been designing -- personal calendars and public "resource scheduling" calendars.

The personal calendar can be used for keeping track of appointments, meetings, holidays, etc. The calendar can be displayed in several ways showing either a summary of the week, a list of appointments on a day, or a diagram of the day showing blocks of free and reserved time. The main operations are "appt" to make an appointment, "cancel" to cancel one, and various display commands. One can attempt to make an appointment at any time. If there is a conflict with another appointment, the calendar reports this fact. If not, the appointment will be made. Appointments are recorded at a particular time with a few keywords to indicate the purpose.

The Conference Room Calendar is similar to the personal calendar in that time slots can be reserved and cancelled. This program is meant to support the reserving of time in one of our conference rooms in the laboratory. The room is generally used for seminars and may involve the coordination of several people and resources. Since a seminar generally has a host who is responsible for the reservation, the host's name is listed in the calendar display as the keyword for the appointment. In addition, there is a form on file for each appointment. The form contains information

about the seminar such as the speaker's name, the title of his talk and whether there will be refreshments. These forms can be active, in which case they may trigger communication with other calendars (such as, the calendar for the person who sets up the coffee pot in time for scheduled refreshments).

7.2. Meetings

A personal calendar can try to call a meeting. The desired length of the meeting, a set of possible times and a list of participants must be specified in the request. The calendar system will try to find a time at which the meeting can be held and will then notify all participants.

For meetings that are called very far in advance of the time at which they will be held, the meeting can be considered to be tentatively scheduled. A scheduler will keep track of several possible times at which the meeting can be held. A second meeting is considered to conflict only if scheduling it (and therefore, removing its time slot from the set of times tentatively reserved for the original meeting) would reduce the set of possible times to less than one. If the second meeting is scheduled, the set of available times for the first meeting is simply reduced. Shortly before the date of the meeting a single time is chosen for the meeting. This can occur either at a "commit" time specified in the call for the meeting or by an explicit request to commit. A caller could specify that he wants a meeting the week of March 10th and that it should be definitely scheduled by March 3rd. Thus the caller can be sure that the meeting will appear on his calendar with sufficient advance notice for planning. If the meeting is committed to a single time too soon, it is quite likely that some participant will have to cancel in order to meet a higher priority commitment that arises later. This would require rescheduling, rather than the simple reduction in the set of tentative times.

7.3. Calendars in a Distributed System

Facilities for coordinating a set of calendars are of use in either a centralized or a distributed system. If the system is to be distributed, its implementation will certainly differ from the implementation of a centralized version. We are assuming that in order to coordinate with another calendar a request must be sent to that calendar. That is, there is no central data base that contains information on all calendars and that can be accessed directly by any calendar.

Operations other than calls for meetings may depend on data at more than one node. For example, when there are tentative meetings (as described in section 2.3) then while a meeting is "uncommitted" the status of certain time slots on the personal calendars of the participants may depend on the status of the tentative

meeting. Thus even if the personal calendars store their data locally, they may have to communicate with the tentative meeting in order to find out whether a particular time slot is free. This can cause noticeable delays if a user is at the terminal trying to schedule an appointment in real time. It also raises a question as to how to display the calendar--should all tentative times for various meetings be shown or should the display show a possible schedule based on information available locally?

Other questions arise:

- How do these data dependencies relate to the dependencies which arise in supporting modular atomic transactions [2]? Are such dependencies at the application level likely to occur in many applications? If so, how can we support their implementation in a programming language for distributed applications?

- Should the caller of the meeting act as the source of information about the tentative meeting? If the tentative meetings are distributed in this way how will scheduling be done if one person is invited to several meetings? Should a central scheduler be invoked to manage meetings? (This latter approach is being explored by an Undergraduate Research Opportunities Program student.)

- Should chains of tentative meetings be schedulable? (E.g., Can I schedule Meeting A conditionally depending on the final timing of Meeting B?) This may save the time of checking with the tentative meeting about a particular time slot. But then how will the system help me in backing out of meetings when conflicts are later confirmed?

7.4. Progress

We have implemented several versions of the calendar. A working version of a single user is available on XX. Draft descriptions of the calendars have been proposed in an internal working paper. A first version of tentative meetings in multi-user calendar system has almost been completed by a UROP student, Pat O'Donnell. The user interface has been studied and a version implemented by an undergraduate thesis student, Eli Wylen.

8. MISCELLANEOUS

8.1. Research in Object Oriented Systems

We have claimed that effective development of distributed system semantics is strongly enhanced by the object oriented view of systems and languages; the view that makes the language or system directly aware of the potentially small storage units which hold the individual data items of relevance to the programmer. Allen Luniewski, in a Ph.D. dissertation, has explored a machine architecture which directly supports this small object view of data management. His thesis suggests that it is possible to provide a reasonable implementation of an object oriented machine, in a manner independent of a particular programming language. In particular, he has demonstrated an architecture that potentially permits objects defined in different languages to be exchanged. In particular, compile time typesafe languages and runtime typesafe languages could presumably coexist in his environment.

8.2. Miscellaneous Distributed System Techniques

Andy Mendelsohn has been investigating the distributed implementation of interactive programs. One example is a distributed editor, with the functions of the editor distributed between a "front-end" personal compiler with highly interactive input and output and a "back-end" compiler such as a timesharing system with higher performance and more storage. The goal is to develop general techniques for distributing functions in any application between a highly innovative front-end and the other compilers in the network. The major accomplishment this year has been the design of a distributed text buffer.

References

1. Reed, D. P. "Implementing atomic actions on decentralized data," Seventh Symposium on Operating Systems Principles, Pacific Grove, Ca., December 1979, 66-74.

2. Reed, D.P. "Naming and synchronization in a decentralized computer system," MIT/LCS/TR-205, MIT, Laboratory for Computer Science, Cambridge, Ma., September 1978.

3. Needham, R. M. and Schroeder, M. D. "Using encryption for authentication in large networks of computer," Communications ACM 21, 12 (December 1978), 993-997.

Publications

1. Luniewski, A. "The architecture of an object based personal computer," MIT/LCS/TR-232, MIT, Laboratory for Computer Science, Cambridge, Ma., January 1980.

2. Marcum, A. "A manager for named, permanent object," MIT/LCS/TM-162, MIT, Laboratory for Computer Science, Cambridge, Ma., April 1980.

3. Saltzer, J. "Environment considerations for campus-wide networks," DARPA Internet Experiment Note IEN-143, March 1980. Available from SRI International, Menlo Park, Ca. 94025.

4. Saltzer, J. "Source routing for campus-wide internet transport," DARPA Internet Experiment Note IEN-144, March 1980. Available from SRI International, Menlo Park, Ca. 94025.

5. Sollins, K. "The TFTP specification," DARPA Internet Experiment Note IEN-133, January 1980. Available from SRI International, Menlo Park, Ca. 94025.

6. Stark, E. "Semaphore primitives and starvation-free mutual exclusion," MIT/LCS/TM-158, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

7. Svobodova, L. and Clark, D. "Design of distributed systems supporting local autonomy," IEEE COMPCON Spring '80, San Francisco, February 1980, 438-444.

Theses Completed

1. Ames, W. "A local area network simulator," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., September 1979.

2. Finseth, C. "Theory and practice of text editors," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

3. Goldberg, D. "A character oriented display editor," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

4. Khalsa, K. "Flow control algorithms for file transfer protocols," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

5. Krueger, S. "System features to aid in the on-line diagnosis of computer peripherals," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980 (also S.B. degree).

6. Leckband, C. "A design of reliability mechanisms for defense minicomputer systems," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

7. Luniewski, A. "The architecture of an object based personal computer," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., January 1980.

8. Myers, B. "Displaying data structures for interactive debugging," E.E. and S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

9. Peterson, H. "Design of source quench congestion control algorithms in interconnected networks," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

10. Pettinato, S. "A sports scheduling system," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

11. Reuveni, A. "The event based language and its multiple processor implementations," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., November 1979.

12. Seaquist, C. "Semantics of synchronization," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

13. Smith, M. "An internet implementation of a terminal access protocol for Multics," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

14. Stark, E. "Semaphore primitives and starvation-free mutual exclusion," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., January 1980.

15. Stathis, G. "A computer controlled telephone dialer," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., January 1980.

16. Toner, S. "Dynamic message routing in interconnected local area data networks," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., August 1979.

17. Vieraitis, R. "Evaluation of the performance of a local area network," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

18. Wylen, E. "A personal calendar: the human-computer interface," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

Theses in Progress

1. Arens, G. "Recovery of a repository in a distributed data storage system," M.S. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

2. Baldwin, R. "An evaluation of the recursive machine architecture," M.S. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1980.

3. Kent, S. "Implementing external protected subsystems in small computers," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

4. Ketelboeter, V. "Forward recovery in distributed systems," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

5. Mendelsohn, A. "Tools for building user interfaces to distributed systems," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

6. Schiffenbauer, R. "Debugging in a distributed system," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

7. Simpson, G. "A monitoring station for a local area network," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected November 1980 (also S.B. degree).

8. Thomas, J. "A multi-protocol network mail transport facility for Multics," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

Conference Participation

1. Corbato, F.J., Received the AFIPS 1980 Harry Goode Memorial Award, at the National Computer Conference, Anaheim, Ca., May 19-22, 1980.

2. Reed, D. "Implementing atomic actions on decentralized data," ACM Seventh Symposium on Operating Systems Principles, Asilomar, Ca., December 1979.

3. Saltzer, J. "Distributed Systems," invited commentary at ACM Seventh Symposium on Operating Systems Principles, Asilomar, Ca., December 1979.

4. Saltzer, J., Summer Study on Air Force Computer Security, Draper Laboratory, Cambridge, Ma., June/July 1979.

5. Svobodova, L. "Reliable distributed systems," IFIP Working Conference

on Reliable Computing and Fault-Tolerance in the 1980's, London, England, September 1979.

Talks

1. Chiappa, N. "The MIT LCS network," Data Communications Group of London of the British Computer Society, University College London, London, England, September 17, 1980.

2. Clark, D. "Local area networks," Greater Boston Chapter of the ACM, Boston, Ma., April 24, 1980.

3. Corbato, F.J. "An overview of computer science research at MIT,"

   MIT-ILP program for several companies, Tokyo, Japan, July 9, 1979;
   Tsinghua University, Peking, China, June 27, 1979.

4. Corbato, F.J. "A management view of the Multics system development," MIT-ILP program for NTTPC, Tokyo, Japan, July 10, 1979.

5. Kent, S. "Implementing external protected subsystems in small computers," IBM San Jose Research Laboratory, San Jose, Ca., March 24, 1980.

6. Mendelsohn, A. "Tools for building user interfaces in a distributed processing environment," Hewlett-Packard Computer Laboratory, Palo Alto, Ca., April 23, 1980.

7. Reed, D. "Using naming for synchronizing access to decentralized data," University of Rochester, Rochester, NY, October 1979.

8. Reed, D. "Implementing atomic actions on decentralized data," Digital Equipment Corporation Research, Maynard, Ma., November 1979.

9. Reed, D. "The distributed data storage system," IBM San Jose Research Laboratory, San Jose, Ca., December 1979.

10. Saltzer, J. "The impact of modern technology on system design,"

    Indian Institute of Technology, Delhi, January 16, 1980;
    Indian Institute of Technology, Kanpur, January 18, 1980;
    Indian Institute of Technology, Madras, January 30, 1980;
    Indian Institute of Technology, Bangalore, January 29, 1980;
    Computing Society of India, Hyderabad, January 31, 1980.

11. Saltzer, J. "Workshop on distributed systems," organizer and lecturer, President Hotel, Bombay, India, January 21-25, 1980.

12. Svobodova, L. "Operating systems for distributed computing," Technical Vitality Program, State University of New York, Binghamton, NY, November 1979.

13. Svobodova, L. "Modeling and semantics of distributed computation," Technical Vitality Program, State University of New York, Binghamton, NY, November 1979.

14. Svobodova, L. "Distributed storage system for a local network," PRIME Computer, Framingham, Ma., June 1980.

Committee Memberships

Chiappa, J.N., DARPA IPTO Internet TCP Working Group

Clark, D.D., DARPA IPTO Internet TCP Working Group

Greif, I., Program Committee for Principles of Programing Languages

Reed, D.P., DARPA IPTO Internet TCP Working Group

Saltzer, J.H., Draper Laboratory Committee on 1979 Security Workshop

Saltzer, J.H., DoD/DDRE Security Working Group Member

DATABASE SYSTEMS

Academic Staff

M. Hammer, Group Leader

Graduate Students

B. Berkowitz
A. Chan
B. Niamir
S. Sarin
S. Zdonik

Support Staff

V. Chambers
S. Sheldon

Visiting Faculty

D. Hsiao

DATABASE SYSTEMS

1. INTRODUCTION

Our activities in database management have ranged over a wide variety of topics, including automatic database design, knowledge-based query processing, distributed database system reliability, application-oriented programming language design, and database computer architecture.

2. IMPLEMENTATION OF A PROTOTYPE PHYSICAL DATABASE DESIGN FACILITY

A. Chan has been engaged in the development of a prototype physical database design facility that embodies the systematic design methodology that we have previously described. This design facility accepts as its input the semantically oriented conceptual schema of an integrated database, a statistical description of its internal characteristics including various size and cardinality information, and a non-procedural description of the repetitive types of transactions to be performed against the database together with their projected frequencies of occurrence and volumes of data accessed. The design system then synthesizes from a wide repertoire of storage and access techniques a good representation for the database to match this projected pattern of use. We envision that such a facility would be an integral part of the next generation of intelligent database management systems that will periodically retune the representation for a database based on observations of the prevailing access requirements. A more immediate application may be towards the design of databases to be implemented on conventional database management systems. In this context, the database administrator will have the responsibility of acquiring accurate statistical parameters required as input to the design facility, though this information may not be easy to come by.

The description of the conceptual schema used by the design facility is in terms of a data model based on Semantic Data Model (SDM). The basic modelling concepts include those of objects and attributes. Objects are grouped into classes based on their types. Objects in the same class possess the same set of generic properties (called attributes), each of which may be single-valued or multi-valued. The values for an attribute may be simple, ranging over an atomic value set that is either enumerated or defined in terms of restrictions of fundamental data types; or complex, ranging over objects in another database class. Object classes can thus be interrelated by complex (reference) attributes. In general, two reference attributes (where one is the inverse of the other) that together describe a binary relationship

between objects in two classes can be declared to be co-primitive; alternatively, only one of them might be designated as primitive while the other is specified as derived by an inversion operation from the other. The significance of this is that only primitive or co-primitive attributes may be directly updated. (Besides inversion, several other standard mechanisms for attribute derivation are also available.) One important modelling concept included in the data model is that of a generalization hierarchy. For example, the class of students and the class of instructors may be immediate descendants of the class of persons in such a hierarchy. An object in the student class represents a real world entity that is both a person and a student. As such, it bears a one to one relationship with the object in the persons class representing the same real world entity. It conceptually inherits all the latter's attributes and possesses as well additional attributes that are specific to a student.

We are assuming that the database is to be stored on mass storage devices that are block oriented. Two important organizational considerations in this context are the provision of efficient access paths to the representation of often accessed objects, and the appropriate intra-and inter-object class clustering of information that is often needed together. Each object in the database may be represented by one or more stored records. A unique identifier is associated with each database object and can be used for efficiently accessing the representation for it. Reference attributes will, therefore, be represented via such identifiers rather than by duplication of all the information in the referenced objects. (The database key of the stored record representing an object can be used as such an identifier. In the event that an object is represented by more than one storage record, then one of these will be designated as the primary record and its database key can be used.) One important clustering issue that the design facility considers is the horizontal partitioning of objects in an object class based on their values for a specialization attribute (i.e., the one that indicates to which descendant class in the generalization hierarchy the real world entity represented by this object belongs). For example, each person may have such an attribute indicating whether he is a student, an instructor, or neither. Objects with the same value for the specialization attribute in such a horizontal partitioning scheme would be represented uniformly by one or more types of storage records. (The attributes stored in different record types are assumed to be non-overlapping.) Each stored record type may be placed in its own linear address space, or multiple record types representing related object classes may be placed in the same address space, with related records clustered together. (Our assumption is that only primary record types will be considered for clustering.) The clustering may be contiguous, with the related records stored adjacently in an hierarchical order, or it may be non-contiguous in which case the related records are only guaranteed to be stored within a small region of the designated address space. In general, we assume that all secondary records will be accessed via the primary record, and that explicit linking pointers from the primary record to each of the

secondary records will have to be stored. In the special case where both the primary record type and a secondary record type are of fixed length, and where each is stored in its own private address space, then implicit linking may be used. (This is done by positioning the two implicitly linked records that represent the same object in the same relative position within the individual address spaces designated for each.) For each record type there is the issue of choosing a placement strategy. This may be *via* that of a related record type; keyed on some attribute or attributes combination; or a non-keyed heap organization can be used. To provide additional access paths to the records, selectors on individual simple attribute values and indices on all values for a simple attribute are considered. To avoid unnecessary complexity, we assume that all pointers involved will point to primary records only. Finally, we also consider the selective explicit maintenance of different types of derived attributes to speed up access to them.

For the purpose of describing the pattern of use of a database, we have focused on devising a non-procedural internal representation of the data requirements of each transaction type and have not concerned ourselves with the actual syntax that would be used at the user interface level. This internal description includes the delineation of the object classes that are involved, their hierarchical relationships as perceived by the application, the criteria for the selection of objects from the different classes, the attributes from each of the selected objects that are of interest, and the desired ordering and nesting requirements on the selected information. Our conception is that there is a single focal object class in each transaction that serves to tie together all of the classes referenced in the transaction. The objects of interest from the focal object class may be selected based on local properties only, or the selection criterion may involve properties derived from other related object classes as well. Once desired objects in the focal object class have been identified, access to all of their primitive or derived properties can then be made. The classes involved in a transaction may be viewed as nodes of a tree, with the root node representing the focal object class. Each node may or may not have restriction predicates attached to the underlying class. The interpretation is that each restricted non-root node existentially qualifies the selection of objects from its parent node. While an unrestricted node is used only for the purpose of "projection," a restricted node may be used both for selection as well as for subsequent projection purposes.

Our use of a non-procedural specification for the requirements of transactions has been motivated by the desire to postpone the determination of processing strategies for them until knowledge of the global requirements of all transaction types has been ascertained, and an appropriate overall organization has been synthesized. However, for any candidate physical organization, there will be many possible strategies for processing such a non-procedurally specified transaction. In order to assign a figure of merit to a candidate organization, part of which involves summing over the processing costs for the different transaction types in the projected usage

pattern, it is necessary to take into consideration the database system's choice of processing strategy for each transaction. For transactions that access objects from multiple classes, two important considerations are the order in which objects in the different classes are accessed, and the method in which the references from one class to another are resolved; in relational terminology the joining order and joining method. (We will adopt the relational terminology in the following discussions for the sake of conciseness.) Since the number of joining orders grows factorially with the number of joins to be performed, it will often be too costly to comparatively evaluate all possibilities. Instead, we assume that the DBMS will first use heuristics to prune the number of candidates for comparative evaluation. We restrict our consideration to join orders that will result in the immediate use of an intermediate result from a previous join in the following join. We also only consider join orders that commence from a restricted node in the query tree. When there are a number of possible joins to be done at the next step, we always take preference on those that involve restricted nodes. Two basic joining methods are considered: a nested (pipelined) method versus a merging (batched) method. The applicability of these methods will depend on the nature of the join to be performed (whether the relationship is one to one, one to many, many to one, or m to n) and how the intermediate results are ultimately required to be sorted and nested. An important component of the comparative evaluation procedure is the ability to combine together the selectivity on different nodes of the query tree. This will facilitate the estimation of costs for sorting intermediate results to facilitate the next merging join, or the computation of the number of lower level operations into which the next nested join will translate. The estimation process makes use of selectivity information on each individual predicate attached to nodes in the tree as well as the combined selectivity of all predicates attached to the same node. In general, when no additional information is available, we assume a uniform distribution of data values and assume that selectivities can be combined by multiplication. However, provision is also made for the specification of the actual average fanout of the hierarchy of objects being selected to permit the more accurate estimation of costs. The prototype design facility is implemented in MDL. The heuristic design procedure used can be conceptualized as having an initialization phase followed by four sequential design phases. In the initialization phase, a default representation is first assigned to each of the object classes. Each is to be represented by a single record type to be stored separately in its own address space, i.e., no horizontal partitioning, vertical partitioning or inter object class clustering will be used in this initial organization. The primary organization for each record type will be keyed on a unique identifier attribute combination for concrete object classes and on a temporal attribute for point event classes. Specialization classes will be keyed on the reference attribute to the related object in the parent class. All inversion attributes are assumed to be explicitly represented. If there are multiple identifier attributes or identifier attribute combinations, then those that are not used as basis for primary organization will be

provided with scatter tables (unique indices). No additional secondary indices will be used for other attributes. It is assumed that inherited attributes in specialization classes will not be explicitly represented, and that the internal representation of the transaction types will explicitly identify all of the classes to which the referred attributes are local. Thus, the notion of attribute inheritance is only used at the conceptual or user level. The transaction types are decomposed into single object class transaction types that may specify the selection of objects from a single class based on primitive attributes, the desired attributes of interest, and the output ordering desired. The selection predicates on each object classes consist of conjunctions of atomic predicates. For predicates involving single valued attributes, equality, range and one of comparators may be used. For predicates involving multi-valued attributes, set comparators may be used. (In fact, we assume that the selectivity of each of these predicates and the average number of attribute values specified are known, and the knowledge by the design system of which set comparator is used is consequently immaterial.) In some cases, the selection predicate may involve a constant rather than a parameterized value. This additional information is only used in the case where the predicate involves a single valued attribute and an equality comparator.

Based on the decomposed usage pattern (in the first phase of the design process), a good representation for each object class is selected in a decoupled fashion. Here the issues of primary organization selection, selector and index selection, and attribute partitioning are considered. An incremental and iterative strategy is used. That is, starting from an initial organization in which no indices or selectors are maintained and in which attributes are not partitioned, successively improved organizations are sought for and then adopted as the incumbent design. (Our assumption is that as a rule of thumb it will be useful to provide efficient access paths for selections based on identifier attribute combinations. Therefore, a scatter table (unique index) will be provided for each set of identifier attributes, unless superseded by the selection of a primary organization based on the same set of attributes.) Instead of assessing the effect of perturbations along different dimensions at the same time, we order the design decisions based on our perception of their relative importance. We consider the choice of primary organization for the representing record type to be of prime importance. Since this can serve to provide a primary access path to the objects based on some aspects of their contents as well as to cluster objects often needed together, an appropriate choice should have the highest potential in achieving good performance. Next, we consider the incremental selection of selectors, followed by the selection of indices, since these are useful in the processing of transactions that access only small fractions of the objects in a class. Finally, we consider the problem of attribute partitioning so as to improve the performance of transactions that access larger fractions of objects in the class, but only a subset of their attributes. We consider this last because we believe

transactions involving a small fraction of the objects are more amenable to optimization, and consequently more sensitive to database organization.

The selection of primary organization involves only the selection of a single structure, and is based on a simple comparative evaluation of the performance of different candidates. In the case of selector and index selection, where multiple structures may be added, a greedy heuristic is used. In each case, instead of considering all possible candidates, potentially profitable ones are identified by examination of the decomposed usage pattern, and then the heuristic is applied. Attribute partitioning is done via a stepwise two-way partitioning heuristic. That is, we start by putting all the attributes in a one-block partition and then proceed to look for a two-block partition that constitutes an improvement over the one-block partition. If this is successful, then further consideration is given to two-way partitioning of one of the two blocks to obtain a three-block partition, etc. To reduce the number of possibilities, an initial grouping of the attributes is considered, which is based on semantic or pragmatic considerations. For example, attributes that together make up a compound attribute are put in the same group, and identifier attributes are placed in the primary record type. In addition, all primitive reference attributes are stored in the primary record type to reduce the level of indirection in accessing information from a referenced object. When the number of resulting groups is still too large, a heuristic clustering scheme is used to obtain a manageable number of groups. (Further discussion of this scheme can be found below.)

In the second design phase, we consider the clustering and partitioning of information in object classes in the same generalization hierarchy. In preparation for this and subsequent design phases, the processing strategy for each type of transaction is reselected in the context of the organization determined by the previous phase. (If it is found to be desirable to index a reference attribute in the first phase, then this is interpreted as explicitly maintaining the inversion attribute in the referenced object class.) It is in this and in subsequent phases that we consider the simultaneous perturbation of the organization of multiple object classes in a restricted and goal-directed way, and the assessment of its effects on transaction processing in a more precise way. However, to keep computation costs manageable, only an incremental and approximate evaluation strategy is used. We assume that the perturbations to be considered in this and subsequent phases will not significantly affect the order in which the joins are to be performed. Therefore, we keep track of the join order selected for each transaction type in the context of the design produced by the first design phase, together with the current cost for processing each of the join. When a perturbation is considered, only the cost of those joins involving the class being perturbed will be recomputed.

In the third design phase, we consider the clustering of information from different

generalization hierarchies. Because of the increased number of possibilities, we attempt to keep the computational efforts manageable by focusing our attention on one portion of the database at a time. We heuristically divide the object classes in the database into groups such that classes in the same group are more frequently used together in transactions. (Classes in the same generalization hierarchy will automatically be put in the same group.) We only consider making simultaneous perturbations to the organizations of object classes within the same group. More specifically, the object class groups are ordered, all perturbations to the organization of one group are made before any perturbations are made to the next group. While a group is being considered, incremental effects of design changes are computed only for those transaction types that refer to classes in this group or in previously considered groups. This phase is begun by considering noncontiguous clustering. It is only when no more noncontiguous clustering can profitably be introduced that converting a noncontiguous clustering scheme to a clustering one in search for further improvement is considered.

The basic grouping strategy involves the definition of a heuristic pairwise similarity measure between object classes that reflects the likelihood of the two object classes being accessed together in an average transaction. A heuristic clustering algorithm is used to permute the similarity matrix to put it into semi-block diagonal form, such that object classes represented by nearby rows and columns of the resultant matrix are often used together. Effectively, the clustering algorithm produces a linear ordering of the object classes. Groups can be formed by identifying the break points in this linear order. Since the purpose of the grouping is to achieve problem decomposition, it is desirable to control the size of the groups being formed. This is done using a sequential partitioning algorithm that minimizes the intergroup similarity measures while keeping the group sizes below a specified maximum.

Finally, in the fourth phase, the introduction of additional redundancies to further improve performance is considered. Recall that some inversion attributes have already been introduced as a result of the phase one selection of an index on a primitive reference attribute; some of these may have been obviated by the clustering introduced in phases two and three. However, it may be profitable to maintain other inversion attributes and also attributes obtained by other forms of derivations.

The debugging, testing and validation of the first version of the design system, incorporating the first three design phases, is being conducted on two different integrated databases. The first is an insurance accounting database with about fifteen object classes, no generalization hierarchies, and fifteen different types of repetitive transactions. The second is a chemical production tracking database with forty-nine object classes and thirty-six different types repetitive transactions. The latter provides a much richer and complex example involving multiple generalization

hierarchies. The transactions here on the average also involve a larger number of object classes. Our strategy for validating designs generated by the prototype system includes careful inspection and analysis, comparisons with designs generated by experienced human designers, as well as analysis of alternative heuristic designs obtained by varying the basic design procedure to cause it to produce multiple designs.

2.1. Query Evaluation System

S. Zdonik has been working on using application domain semantics to improve the performance of a query processing system. We have produced a system architecture for a knowledge-based query processing system and a methodology for creating new query improvement programs that can be easily integrated into this system. Our approach is based on a multiprocessing model of the search process that is guided by a set of heuristics.

In modern query languages, it is easy for an end-user to express queries that would be extremely costly to evaluate if performed in a straightforward, brute force manner. In order to address this problem, we have been investigating a different approach to database query optimization, one that seeks to exploit knowledge about the application domain of the query to transform the original query into a different one, which is equivalent to the original (in the sense of being satisfied by the same set of records from the database) but cheaper to evaluate given the existing database configuration. Such transformations are likely to be effected on the source level version of the query; that is, rather than seeking the most efficient way of processing the query as stated, we are seeking way to restate the query and thereby effectively construct a different request, one which is not syntactically equivalent to the original but does result in the same output because of the semantics of the application.

The query language and the data model on which this system is based view the database as a collection of sets of objects. There are functions that map objects into sets of objects or sets of scalar values. This is similar to entity-relationship approaches to data modeling, and is based on our earlier work on SDM.

The query improvement system is organized around sets of programs called improvement techniques. Each technique is capable of performing a single type of transformation on a given query expression. We have identified several query improvement techniques in this work. However, we feel that the primary result of our work is the framework for the design of such query improvement programs rather than the actual techniques presented in such a list.

An example of a simple technique is one that will be called Domain Refinement.

Domain refinement only works for restriction expressions; a restriction is a function that applies a predicate to each member of a set (called the domain of the restriction). The value of the restriction is the set of elements of the domain that satisfy the predicate. In domain refinement, a restriction of a large domain is converted to a restriction of a smaller domain for which the elements are easy to materialize.

Consider the query *Find all employees who make more than $40K*. This can be transformed to *Find all employees who are managers and who make more than 40K* if the following facts are known.

1. The employees who make more than 40K are a subset of the employees who have the job of manager.
2. The set of employees is indexed on the attribute "job," one value of which is manager.
3. The size of the set of employees is 10,000.
4. The size of the set of managers is 100.

The first fact makes it possible to perform the transformation since only then will the meaning be unchanged. The other three facts make the new query more desirable than the original query, since the total number of records that will need to be individually retrieved is smaller.

The problem that must be addressed in effecting such transformations against a query in the context of a large knowledge base is coping with the combinatorial explosion of versions of the query that result from applications of sequences of transformations. Our approach to this problem is based on a multiprocessing model with an heuristic scheduler. Each kind of transformation that might be applicable at a given stage is treated as a process, with the scheduling of that process based on the likelihood of the transformation succeeding. A system architecture embodying this approach has been designed and is now being implemented.

2.2. Reliability Mechanisms in Distributed Information Systems

In recent years, a considerable amount of research has been done in developing techniques for ensuring data consistency and correct operation of a distributed information system in the face of both conflicting concurrent operations on the data and asynchronous failures and recoveries of system components. This research has yielded important results, such as the "two-phase commit" protocol. However, most of the proposed techniques suffer from the problem that the failure of a single site can render data at other sites unusable for an indefinite amount of time, until the site in question recovers. This is unacceptable for systems in which it is important to provide a reasonable response time, and alternative solutions are necessary.

The techniques used to resolve the above problems can be broadly classified into two types: *passive* and *active*. Passive reliability techniques distribute the responsibility for a transaction among multiple sites and use voting to resolve the transaction's final outcome (i.e., to "commit" or to "abort"). Examples of this are provided by Thomas's majority consensus algorithm and Reed's multisite "commit records." It can be correctly argued that the use of voting makes these methods resilient to multiple site failures, and that response time can be kept within reasonable bounds if votes are collected in parallel. However, distributed voting techniques seem to incur a considerable bookkeeping overhead, in terms of the large number of messages required to collect individual votes in order to arrive at a decision, and to then reliably inform all voters of the final outcome so that each may reclaim the storage allocated to keeping track of the transaction.

Whereas passive techniques use redundancy and voting to protect against failures, the main thrust of active reliability techniques is to take positive action in response to failures. This assumes the ability to detect failures, which in turn requires some notion of physical time in order to distinguish between failures and mere delays. Thus, active techniques are generally based on the ubiquitous strategy of "timeouts."

The most ambitious example of the use of active reliability techniques in distributed systems is provided by Hammer and Shipman's work in SDD-1. A "Global Time" subsystem (or "layer") is described that presents the illusion (to the application) that each site is either "up" or "down" at every instant of time with respect to a global network clock. Application programs can then be written to exploit the notions of site status and global time. In particular, algorithms have been developed for reliable message delivery, atomic broadcast of multiple updates, and reliable synchronization against conflicting transaction classes; these algorithms can be made resilient to any desired number of site failures.

An investigation of the techniques used in the SDD-1 global time subsystem has

been carried out by Sunil Sarin, in order to determine the conditions under which correct operation of the subsystem can be ensured. The global time subsystem does use the above mentioned strategy of timing out a remote site on its response to a message, with an additional mechanism, the "YouAreDown" message, for crashing a site that is slow to respond. A critical requirement was identified here, that a YouAreDown message must be guaranteed to have effect (i.e., reach the remote site and cause it to crash in response) within *bounded time*, and that a site sending a YouAreDown message must *wait* the proper amount of time before asserting that the other site is in fact down. If this cannot be ensured, it is possible for the system to reach a state in which two sites are up but believe each other to be down; while this state might not persist for long, it is possible for application processes at the two sites to perform actions that are inconsistent with each other and hard to rectify after the fact.

Various site recovery protocols and clock synchronization techniques, both those used in SDD-1 and alternatives, were also investigated, yielding different wait times in detecting site failures and recoveries. The main results of the study have to do with the applicability of the notions of global time and explicit site status to distributed information systems. First, these notions are useful only if time is an explicit part of the application, as in timestamp-based systems such as SDD-1. The increasing frequency with which distributed algorithms that use timestamps appear in the literature may serve to indicate that a global time subsystem would in fact be generally useful. However, the utility of these notions is limited by the feasibility of correct implementation. It is well nigh impossible to guarantee an acceptably small message delay (which is necessary for "YouAreDown" messages and the like) in a long-distance packet-switched network with multiple applications contending for resources at time-shared host nodes (such as the Arpanet). On the other hand, it ought to be more feasible to implement a global time subsystem in a closely-coupled multiprocessor network dedicated to a small number of application functions; this issue is currently under investigation.

2.3. DIAL

Brian Berkowitz has completed the design of a database programming language called DIAL that is intended to be used for coding data intensive applications systems (such as purchasing systems, accounting systems, and reservation systems). The conventional approach to implementing an application system is to augment a general purpose programming language to allow it to employ a data management system to obtain and modify values from a database. Our approach is to develop an integrated facility that directly incorporates data management capabilities into a programming language, and moreover, includes in the language those constructs most useful for coding database application program. The design

of the language has been driven by the analysis of a number of application systems. Moreover, we have coded two complete systems, a purchasing system (which is an example of a transaction based system) and a job-scheduling system (an example of a batch oriented application involving complex computation and correlations of data). (We have also used these systems to evaluate our language according to criteria such as brevity.) In studying these systems, we have identified a number of frequently used program constructs and have incorporated them into the language as high-level applications specific facilities.

The data model employed by the language (i.e., the mechanism used to describe the data used by an application system) is derived from SDM. The basic concept of this model is the entity which represents an object of interest in the application. Entities are organized into classes, and are described by means of attributes; all entities in the same class have the same descriptive attributes. Every attribute of an entity has a value which is drawn from a specified value class. In general, the value class of an attribute will be an entity class or a set of primitive values (such as integers, strings, etc.). An attribute may be either single or multi-valued, and may be optional or mandatory. (A mandatory attribute must have a non-null value, and so must be initialized when the entity is created.) A more fundamental difference is between primitive and derived attributes. A primitive attribute is one that is subject to direct initialization and updating by operations invoked in a procedure. A derived attribute is associated with an entity but is automatically computed from other information in the database. Thus the definition of a derived attribute specifies the way in which it will be computed; every time the value of the attribute is used in a procedure, this definition is implicitly employed to derive the value. Thus, the value of a derived attribute may not be explicitly changed by any procedure.

The full range of DIAL expressions may be used to specify the value of a derived attribute. DIAL provides a relatively standard repertoire of scalar and aggregate operators that can be applied to primitive values, as well as special operators on classes that produce new classes. (Thus, a class-valued expression would be used to specify a class-valued derived attribute.) A class may be constructed in a number of different ways including applying a Boolean operator to two classes (e.g., union or intersection) and restricting a class on a predicate.

A DIAL procedure can directly access the database of an applications system. Procedures can take entities and aggregates of entities as arguments and return them as results. High level operators can be used within procedures to manipulate aggregates of entities (e.g., restriction, union, and intersection). The same types of expressions that are used to define derived attributes may be used to perform computations in procedures.

A number of special purpose control structures are provided by the language.

These include a variety of conditional statements and a sophisticated exception handling facility, similar to that found in CLU. Several important specialized types of procedures are included. These procedures allow for specialized types of control that cannot be easily simulated using normal procedures. Two examples of these special procedures are:

- Case – procedures: these provide the ability to dispatch to a number of different procedures based on the class of an entity provided as an argument, on the value of some expression computed from the arguments, or on a set of predicates.

- Iterators: these provide a mechanism for creating an aggregate. The iterator contains a loop which in turn contains yield statements that identify entities that are to form a new class returned by the iterator.

A special mechanism called a *controller* is provided to allow program specific resolution of synchronization conflicts that arise when several concurrently running users attempt to update the same data. This facility provides more flexibility than is possible in most database systems that employ automatic abort and restart facilities for resolving synchronization conflicts. A controller is similar to a monitor; it provides a set of operations and has a local database used by these operations in order to make synchronization decisions. Normally only one transaction can interact with the controller at a time; this serialization is used to provide synchronization. The controller has the ability to abort transactions that have invoked it, which may be necessary to prevent deadlock.

In the past year we have included an additional facility in DIAL that allows user-computer interactions to be specified in a high level manner. User-computer interactions play a major role in many transaction based application systems. Users of a system often provide input by interactively completing a variety of specialized forms that are displayed on the screen, and the system displays information in response to user inquiries. In current practice, facilities for conducting these user-computer dialogues are programmed in conventional languages. The various mechanisms for laying out the screen, handling data entry, verifying that input data is correct, and modifying the course of the dialogue in response to user input, all are coded in a procedural fashion. The resulting code is often extremely long and very tedious to produce and debug.

The current version of DIAL embodies a simple model of the structure of these dialogues. At appropriate times, a form will be displayed to the user; a form consists of a number of rectangular areas in which data values may be displayed or entered. Some forms are used purely for data display, others for data entry, and others for both purposes. Various labels and prompts may be displayed on the form, including

menus from which the user is to select an item. When the user wishes to enter data, he employs whatever positioning device provided by his terminal to move to the appropriate area; DIAL programs do not impose a fixed order in which a form must be completed. Dynamic dialogues, in which the user's actions determine what additional information is to be displayed, can also be supported.

An object, called a *port*, provides an abstract interface between a DIAL program and the terminal. A port is very similar to an entity; it is described by attributes. Rather than being stored in a database, the values of these attributes are displayed to or entered by the user. Ports are characterized by port type, the definition of which identifies the attribute definitions together with all the information necessary for conducting a user-computer dialogue (screen layout, input and output formats, etc.). A program initiates a user-computer dialogue by creating a port of a specified type and setting the values of some of its attributes. The port type provides the information used by the DIAL run-time environment to automatically conduct the dialogue. Upon completion of the interaction, the program can obtain the values of those attributes whose values were entered by the user.

There are two interesting aspects of the mechanism just described. The first is that "knowledge" about how the dialogue is conducted is declaratively expressed in the port type definition. This is in keeping with our overall goal of reducing the scope and complexity of the procedural portion of an application system. The second aspect is that the facilities for manipulating ports are very similar to those for manipulating entities. The program interacts with both these types of objects by examining and updating attributes. As far as the program is concerned, a port is essentially an entity, the data being supplied from the "external" environment rather than from the database. This uniformity simplifies programming by reducing the number of new concepts needed in order to use ports; it also provides for a limited form of "information hiding."

DIAL differs from other database languages in a number of important respects. Unlike other database languages that simply integrate DBMS facilities into a conventional programming language, the design of DIAL was driven by understanding applications. This resulted in a language that uses a high level data model instead of the more conventional relational or network models. DIAL includes a parameterized subsystem for specifying user-computer dialogues. DIAL also allows program control over the resolution of synchronization conflicts.

2.4. DATABASE COMPUTER ARCHITECTURE

Professor David Hsiao of the Ohio State University has been a visitor in our group this year and has focused on the impact of very large databases on database

computer (hardware) architecture. To this end, the study has produced three findings, which are briefly outlined in the following paragraphs; all of these findings have been written up as papers.

The first finding on the impact of large databases on database computer (hardware) architecture is that the new storage requirements and transaction execution times for a very large database on a database computer may not exhibit uniform improvements over the old storage requirements and transaction execution times for the same database running on a conventional computer with traditional database management software. The lack of storage savings in the new architectural environment is due to the fact that the replacement of the address pointers of the old physical structure with the symbolic descriptors of the new normalized structure tends to increase the storage requirement in the new environment. On the other hand, the built-in parallelism and specialized hardware for database management tend to improve the transaction execution time greatly. Upon closer examination, these performance gains are hindered in many cases by the traditional way in which the transactions are prepared. The insistence on one-record-at-a-time operations and on the sequential access of large amounts of data has rendered the advanced features of new database computers less effective. The study concludes by suggesting that transactions should be implemented predominantly with all-records-at-a-time operations and the content-addressing of large amounts of data, in order to capitalize on the advanced features of the new database computers.

The second finding is that the auxiliary information and update operations of a large database may impose severe penalties on the cost and performance of various database computer organizations. In the absence of separate hardware storage and processing components for auxiliary information about the database, the database computer tends to be organized cellularly, where each cell is required to have some on-board processing logic. Such organizations are deemed to be too costly and complex to be realized in hardware for large databases. By utilizing certain auxiliary information and relegating this information to separate storage and processing components, the database computer organization can be made simpler with lower cost. However, such database computers process update operations in two steps--first, the information in the main database store is updated; then the auxiliary information in the separate components is updated. As a two-step operation, the update creates a phenomenon, known as update blocking. In other words, until the second step of an update operation is completed, subsequent operations cannot be carried out. Consequently, the possibility of a high degree of concurrent operations on the shared database is small. The purpose of blocking other operations is primarily to prevent these other operations from receiving erroneous information from the database or storing information in the wrong place, since these operations depend on the correctness of the auxiliary information being updated. Update

blocking imposes a severe performance penalty, particularly on a database computer of multiple-instruction-and-multiple-data-stream (MIMD) organization. MIMD database computers are meant to carry out concurrent operations. Unfortunately, update blocking may render the potentially high degree of concurrency of an MIMD database computer unrealizable. The study concludes by advocating the single-instruction-and-multiple-data-stream (SIMD) organization for database computers. In addition to the absence of blocking problem, the SIMD database computers incur lower cost and complexity in hardware realization.

The third finding focuses on the design and analysis of hardware sorters. Sorting plays an important role in many database operations, such as the relational join. Consequently, sorters with high performance are desirable. By utilizing processors in parallel, sorters can be designed to achieve this high performance. However, for hardware realization, two limitations must be overcome. First, the interconnecting processor-to-processor communication and control buses must be few. Second, the architecture must be expandable as the size of the database increases and the need for more parallel processors grows. A solution that requires only two interconnecting parts for each processor of a highly expandable and multiprocessor oriented sorter has been proposed; the analysis of the time and space complexity of the sorting algorithm and the comparison with many existing parallel sorting algorithms have been performed.

Publications

1. Hammer, M. and Berkowitz, B. "DIAL: A programming language for data intensive applications," Proceedings of the 1980 SIGMOD International Conference on Management of Data, Santa Monica, Ca., May 1980.

2. Hammer, M. and McLeod, D. "On database management system architecture," MIT/LCS/TM141, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

3. Hammer, M. and McLeod, D. "SDM: A semantic database model," accepted for publication in ACM Transactions on Database Systems.

4. Hammer, M. and Shipman, D. "Reliability mechanisms for SDD-1: A system for distributed databases," accepted for publication in ACM Transactions on Database Systems, May 1980.

5. Hammer, M. and Zdonik, S. "Knowledge based query processing," accepted for publication in Proceedings of the 1980 Very Large Data Bases Conference.

6. Hsiao, D. "Design issues of high-level language database computers," Proceedings of the High-level Language Computer Architecture Workshop, Ft. Lauderdale, Fla., May 1980.

7. Hsiao, D. "The design and analysis of parallel hardware sorters," accepted for publication by the Ohio State University as a technical memorandum.

8. Hsiao, D. "The impact of the auxiliary information and update operations on database computers," accepted for publication in the Proceedings of the International Congress on Applied Systems Research and Cybernetics, December 1980.

Theses Completed

1. Berkowitz, B. "Design of a language of coding data intensive applications systems," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

2. Zdonik, S. "On the use of domain-specific knowledge in the processing of database queries," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

Theses in Progress

1. Chan, A. "A methodology for automating the physical design of integrated databases," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

2. Niamir, B. "Image and graphics handling in an advanced office workstation," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1981.

3. Sarin, S. "Reliable real-time synchronization in distributed information systems," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1981.

Talks

1. Hammer, M. "The life cycle of a database," 1979 National Computer Conference, New York, N.Y., June 1979.

2. Hammer, M. "Design and implementation of database management systems," Bell Laboratories, Whippany, N.J., August 1979.

3. Hammer, M. "ADA and computer science research,"ADA Test and Evaluation Workshop, Boston, Ma., November 1979.

4. Berkowitz, B. "DIAL: A programming language for data intensive applications," 1980 SIGMOD International Conference on Management of Data, Santa Monica, Ca., May 1980.

5. Chan, A. "A methodology for physical database design,"

   Bell Laboratories, Naperville, Ill., September 1979;
   Computer Corp. of America, Cambridge, Ma., November 1979.

6. Chan, A. "The design and implementation of a prototype physical database design facility,"

   IBM Research Laboratory, San Jose, Ca., December 1979;
   Stanford Research Institute, Berkeley, Ca., December 1979;
   Xerox PARC, Palo Alto, Ca., December 1979.

7. Hsiao, D. "The database computers,"

   The Executive Office of the President, Washington, D.C., January 1980;
   Christian Michelsen Institute, Bergen, Norway, February 1980;
   University of Trondheim, Trondheim, Norway, February 1980;
   University of Paris, Paris, France, February 1980;
   INRIA, Le Chesnay, France, February 1980;
   Workshop on Computer Architecture for Non-numeric Processing, Pacific Grove, Ca., March 1980;
   IBM Santa Teresa Lab, Ca., March 1980;
   Bell Laboratories, Murray Hill, N.J., March 1980;
   Academia Sinica, Peking, China, April 1980;
   Hwa-Chung Engineering Institute, Wuhan, China, April 1980;
   National Computer Conference, Anaheim, Ca., May 1980;
   Applicon Inc., Burlington, Ca., June 1980;
   Digital Equipment Corporation, Merrimack, Ma., June 1980;
   IBM Scientific Center, Cambridge, Ma., June 1980;
   Sperry Research Center, Sudbury, Ma., June 1980.

DIGITAL INFORMATION MECHANICS

Academic Staff

E. Fredkin, Group Leader

Research Staff

T. Toffoli

Undergraduate Students

S. Berez
R. Bernstein
D.N. Chin
R. Fearing
A. Iqbal
A. Ishtiaque
V. Nepustil
W. Pong
S. Ross
G. Vachon
M.A. Wolinsky

Graduate Students

R. Giansiracusa
N. Margolus
D. Payton
A. Ressler

Support Staff

A. Schmitt

DIGITAL INFORMATION MECHANICS

1. CONSERVATIVE LOGIC AND REVERSIBLE COMPUTING

The ultimate goal of conservative logic is dissipationless physical computing. At a microscopic level, all physical processes are strictly reversible and dissipationless. Traditional schemes for physical computation are based on irreversible macroscopic processes. From a technological viewpoint, these processes have been easy to achieve since they do not require very exact control of system parameters, initial conditions, and external disturbances; on the other hand, irreversible computation entails damping and for this very reason it requires energy dissipation in order to overcome thermal noise.

In the past year we have completed the essential theoretical foundations for reversible computing in an abstract context (invertible Boolean primitives, interconnection rules that preserve invertibility, effective design criteria for the elimination of "garbage" information in reversible digital networks, analysis of trade-offs between number of gates, maximum propagation delay, amount of temporary storage, and number of constant inputs and garbage outputs), and we have attacked the problem of translating such abstract principles into physical models.

We have discovered a very interesting physical model of conservative logic based on the elastic collision of identical "balls" (i.e., based on the same stylized physical rules that are at the basis of the kinetic theory of the behavior perfect gases). In this model, the main information processing primitive is the "interaction gate," a two-input, four-output primitive out of which essential higher-level primitives (such as the Fredkin gate and the crossover element) can be synthesized in a very simple way. We have explored many variations of this model and we have analyzed trade-off involving the number of collisions, the number of static reflectors, the number of recirculating balls, and the nature of the collisions themselves.

At present, we are directing our research toward more detailed physical models, and we are trying to reach a deep understanding of the many physical issues involved in the overall computational process (initialization and readout, quantum effects, thermodynamical constraints, isolation from noise, etc.).

2. SEMI-INTELLIGENT CONTROL

Semi-intelligent control is an original approach to the problem of controlling machinery through the use of a distributed network of microprocessors. Our goal is

to identify a basic set of general-purpose capabilities to be given in an identical fashion to each microprocessor of the network, independently of its assigned task. These capabilites include communication, generation and use of look-up tables, handling of sensors and effectors, construction and use of a simplified internal model of the whole network. Based on the guarantee that each node of the network possesses such capabilities, the programming of the network for a given application becomes much easier, faster, and reliable, and can be carried out in terms of high-level constructs that are goal oriented rather than process oriented.

Our current research involves top-down aspects (case studies, such as the Hinge system discussed below) as well as bottom-up aspects (hardware, bus, memory, and I/O architecture, communication protocols, firmware, etc.).

The Hinge system is a typical paradigm for semi-intelligent control studies. It consists of two "sticks" connected by a joint having a single degree of freedom. In an extensive and physically very detailed simulation we have shown that, under appropriate control, this system can balance itself, hop and move about, and recover from disturbances. One of our goals is to develop for this system a set of low-level reflexes based on the intelligent use of sensor information and look-up tables containing in a compressed form a great deal of practical dynamics. Once these reflexes are established at least in a rudimentary form, the system can be programmed for higher-level deambulatory tasks. We are working at a hardware prototype of the Hinge system, where sensors, actuators, and "number-crunching" processors will be controlled by individual microprocessors in the environment of a semi-intelligent control network.

Bottom-up development work on semi-intelligent control has been going on for a number of months on an experimental basis, in anticipation of regular funding. It has involved weekly meetings with a number of students, lab sessions, and many individual lab projects aimed at particular hardware, software, and architecture aspects. The core of this lab activity consists of number of 6801 microprocessors provided with serial and parallel interfaces, sensors and display devices, and other specialized hardware, and interconnected through a broadcast bus (the so-called PSI) bus. Besides studying hardware and protocol specifications for the PSI bus, we have developed a separate "pseudo-ROM" bus, on which the prototype designer can interact with the individual microprocessors and specify their behavior through program and data streams downloaded from a system-development machine (such as ITS or, in the near future, a LISP machine). Other topics of this research have involved experiments in motor control, D/A and A/D conversion, optical coupling, and digital feedback loops.

Publications

1. Fredkin, E. and Toffoli, T. "Conservative logic," (advanced draft).

2. Toffoli, T. "Bicontinuous extensions of invertible combinatorial functions," Mathematical Systems Theory 13, 4 (1980).

3. Toffoli, T. "Reversible computing," MIT/LCS/TM-151, MIT, Laboratory for Computer Science, Cambridge, Ma., February 1980. (To be submitted for publication, and accepted by the 5th Internat. Colloq. on Automata, Languages, and Programming, Holland, July 1980.)

Theses Completed

1. Berez, S.H. "Protocols for microcomputer intercommunication," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

2. Bernstein, R.D. "A low-dissipation rectifier using FET switching," S.B. thesis, MIT, Physics Department, Cambridge, Ma., May 1980.

3. Chin, D.N. "Physical implementations of conservative logic," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

4. Iqbal, A. "Investigation of a serial bus protocol," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

5. Ross, S.I. "State-space trajectory planning, tracking, and optimization," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

6. Vachon, G. "Reversible programming," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., December 1979.

7. Wolinsky, M.A. "Elementary properties of reversible digital difference equations," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., December 1979.

Other Documentation

1. Fredkin, E. and Toffoli, T. Letter Proposal on "Design principles for achieving high-performance submicron digital technologies," (containing a preliminary description of research on the interaction gate).

2. Giansiracusa, R. "A model for evolutionary development of intelligence and its application to a robotic system for learning neuromuscular control," (term paper).

KNOWLEDGE BASED SYSTEMS

Academic Staff

W. A. Martin, Group Leader

Research Staff

G. Brown
G. Burke
E. Golden
L. Hawkinson

Undergraduate Students

F. Hirsch

Graduate Students

G. Faust

Support Staff

A. Schmitt

KNOWLEDGE BASED SYSTEMS

1. INTRODUCTION

Our work over the past year naturally divides into automatic programming, natural language processing, and system support.

2. AUTOMATIC PROGRAMMING

Greg Ruth completed a write-up of the PROTO-SYSTEM I project. This will appear soon as a technical report. Currently, Greg Faust is working on a program to unprogram actual COBOL programs into the very high level language used in PROTO-SYSTEM I. This is done by first passing the COBOL program through an unprogramming system written by Richard Waters for his Ph.D. dissertation. A program written by Glenn Burke reads the COBOL and converts it to the internal form used by Waters as input. Waters has extended his program to handle constructions found in the COBOL programs which were not in the FORTRAN programs he analyzed in his dissertation. The output of Water's program has separate explicit representations for the control flow and data flow of the COBOL program. Faust's program reads Water's output and constructs the very high level language corresponding to the COBOL program. It works primarily by recognizing standard patterns which it re-expresses as high level constructs. If it gets stuck, it enters an interactive mode.

The output of Faust's program can be used to generate a new PL/I program using the PROTO-SYSTEM I program generator.

3. NATURAL LANGUAGE PROCESSING

Professor William Martin worked out a new theory of the logical form of English sentences. This appeared as MIT/LCS/TM-139 and will reappear shortly in revised form.

A new natural language processing system was implemented. This system features an especially fast parser which finds all the syntactic parses of a sentence breadth first. The parser can be described as a variant of Early's algorithm indexed first by state and second by word position. Currently, syntactic, semantic, and pragmatic analyses are performed sequentially, as each is a more computationally expensive operation than the preceding one. However, on long sentences, or

sentences with multiple conjunctions, the number of syntactic parses explodes to the point where pruning of syntactic paths with semantic constraints would probably pay. This is currently being explored. This system is being extended to handle all the sentences encountered by Ashok Malhotra in his experiment with real users.

The semantic component eliminates parses by applying predicability constraints. Predicability is much less restrictive than truth in the world. For example, "meow" is predicable of "dog," even though it is generally false. "Green" applied to idea is an example of a predication which would be rejected by the predicability check. Predicability solves the problem of checking counterfactuals--e.g., "the dog meowed" and "the dog did not meow" are both OK, while "the idea is green" and "the idea is not green" are equally bad. Failure of predicability says that two concepts are so unrelated that their combination cannot even be constructed. Since it is a very general constraint, it is easy to apply, but it proves to be useful.

Since the system implemented has separate components for syntax, semantics, and pragmatics, and since these are implemented in a fairly general way, it has been possible to implement a learning module which can learn new facts needed to handle a new input which falls within the general domain of competence of the system. The general rules in the system make it possible to guess the nature of missing information and obtain this by asking the user multiple choice questions. This allows a user to extend the competence of the system without having to know its internal construction and without having to keep a lot of contextual detail in mind.

Working toward an application of this system, Gretchen Brown partially implemented a data-dictionary dictionary system. This system interactively acquires a semantic data model (of the Hammer/McLeod type) of a number of data bases. The idea is then to answer questions about what data is in what data bases using the above natural language processing system.

4. SYSTEM SUPPORT

Prof. Szolovits and Prof. Martin have implemented an extension to LISP called BRANDX. This extension features:

- property lists on all lists,
- optional labels on all data items, and
- unique and non-unique lists.

The goal of BRANDX is to provide LISP with data objects which are unique with

respect to some of their components. These components are put in the list and the other components are put on the property list of the list. Normally a unique data object is identified by these unique-izing components and so they normally print while the components on the property list don't. Labels are used to reduce the print size of objects and to facilitate the input of circular structure. Circular structure is also facilitated by the use of a "pronoun" by which a subexpression of an S-expression can be the whole expression.

Lowell Hawkinson and Glenn Burke wrote a large package of macros called LSB. This package facilitates the building of large LISP systems. It addresses modularization, documentation, and the setting up of environments at run time and compile time.

Glenn Burke and David Moon implemented a very nice facility, called LOOP, for specifying loops in LISP programs.

Fred Hirsch and David Moon got the Webster's Pocket Dictionary onto our disk and wrote some functions for accessing words and doing studies of word definitions.

Publications

1. Brown, G.P. "Action descriptions in indirect speech acts," Cognition and Brain Theory 4, 3 (Spring 1980).

2. Brown, G.P. "Toward a computational theory of indirect speech acts," MIT/LCS/TR-223, MIT, Laboratory for Computer Science, Cambridge, Ma., September 1979.

3. Martin, W.A. "Roles, co-descriptors, and the formal representation of quantified English expressions," MIT/LCS/TM-139, MIT, Laboratory for Computer Science, Cambridge, Ma., September 1979.

Theses Completed

1. Koton, P. "Simulating a semantic network in LMS," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., December 1979.

PROGRAMMER'S APPRENTICE

Academic Staff

H. E. Shrobe

R. C. Waters

Graduate Students

D. C. Brotsky

G. G. Faust

D. G. Shapiro

R. D. Duffey

C. Rich

Undergraduate Students

D. Chapman

Visitor

C. Frank

PROGRAMMER'S APPRENTICE

Richard Waters is working on a demonstration system which will exhibit some of the capabilities of the proposed Apprentice system. This demonstration system is a plan based editing system. Using the system, a user will be able to modify his program in three ways. First, he can edit it by adding and deleting characters as in an ordinary text editor. Second, he can modify a program by issuing requests based on the PLAN corresponding to the program. For example, a user might say: "add this filter into that loop" or "change the initialization of that summation to (MAX A B)." The key feature of this interaction is that the editor makes available to the user a vocabulary of terms so that he can treat his program as an algorithmic structure, rather than as a mere character string. Third, he can directly edit a graphical representation of the plan for the program.

In order to support the three modes of interaction, the editor maintains three representations of the program being edited. First, it has a representation of the program as text. Second, it has an analyzed plan for the program. Third, it has a graphical representation of this plan. Each kind of editing request modifies one of these forms. The system propagates the changes to the other forms. For example, if the user changes the text, the program is reanalyzed in order to determine what the new plan should be, and if the user specifies a change to be made to the plan, the editor determines what changes this causes in the text.

Most of the pieces needed for this system were constructed prior to last summer. The most important of these is an analyzer which can construct a plan corresponding to a section of program text. Only three pieces remained: a coder module which could take an analyzed plan and produce code corresponding to it; a plan drawing module which could produce the graphical representation corresponding to a plan; and a user interaction module which would allow a user to specify plan based changes in his program. Following the design laid down in his thesis, Waters produced the coder this fall. The plan drawing module is being constructed by Dave Chapman. Waters will construct the user interaction module this spring.

Charles Rich is finishing up his Ph.D. dissertation on representing and cataloging knowledge about basic algorithms and data structures. He expects to be done in June 1980. His thesis focuses on two typical implementations of an associative retrieval data base. The first example program is a simple hash table which uses an array of LISP-style association lists. The second example program stands at the

limits of his current cataloging effort. It is a complicated data base implementation (similar to the one used in early AI programs, such as CONNIVER) which uses ordered lists and a multiple intersection strategy. These programs are analyzed with a hierarchy of descriptions which make explicit the way in which standard programming ideas are applied in these particular programs. The standard programming ideas are formalized as "plans." For example, the Aggregation plan captures the common idea underlying summation, set aggregation, and accumulating objects in a list. An example of a less abstract idea which is formalized as a plan is the use of "flags" in programming. Flags are used to encode information about splitting in control flow which can be recovered later by testing the flag and splitting again. The result of Rich's thesis will be a library of programming plans which has applications in the Programmer's Apprentice for the automated analysis, synthesis and verification of programs.

Richard Waters, Gregory Faust, Glenn Burke, and Professor William Martin have been working on a project to construct a COBOL to HIBOL translator. The system works in four stages. Glenn Burke wrote a module which converts COBOL programs into a LISP-like intermediate form. Waters rewrote the plan analyzer so that it could operate on this intermediate form as well as on ordinary LISP. The plan analyzer mentioned above can then be used to develop an analyzed plan corresponding to the original COBOL program. As his graduate thesis (co-supervised by Martin and Waters) Gregory Faust is writing a program which takes an analyzed plan and creates a HIBOL program corresponding to it. He expects to be finished with this by the end of next summer.

Roger Duffey is investigating the possibility of doing a graduate thesis under Richard Waters. He is thinking about a novel approach to compilation. The key idea is that rather than proceeding via a series of macro expansions, and local optimizations, his system will first analyze the program to obtain an abstract description of the behavior of the program which is then reimplemented using detailed knowledge of how to write efficient assembler language code for a particular target machine. His system would start with an analyzed plan for the program which is then further analyzed. He expects to complete a proposal in the coming spring.

Dan Shapiro is also developing a graduate thesis topic with Richard Waters. He is thinking about constructing a system "Sniffer" which locates bugs in a program. The system is organized as a group of experts each of which knows a lot about a specific bug, and a group of feature detectors which are used to determine which experts should be used in a given situation. The experts and feature detectors rely on two principle sources of information: an analyzed plan for the program being investigated, and a trace of the execution history of the program including complete information about all of the side-effects which occurred. The organization of the

system makes it possible for it to exhibit deep knowledge of a restricted class of bugs, rather than shallow knowledge of bugs in general. He expects to complete the proposal in January 1980.

Dan Brotsky is looking into the possibility of doing a graduate thesis with Charles Rich and Professor Patrick Winston. He is interested in developing a system which will do in depth recognition of algorithmic structures in a program. Waters' analyzer only recognizes a few simple stereotyped forms in a program. Brotsky's analyzer would go beyond this and recognize the kind of detailed structures that are in Rich's plan library.

Claude Frank is visiting with Prof. Gerald Sussman and the Artificial Intelligence laboratory for a year on leave from the Schlumberger Corporation. He is looking into the possibility of doing a project with the Programmer's Apprentice group. He is interested in using the plan for a program as the basis for a simple automatic documentation facility. The central problem in this is deciding what not to say. The plan is very verbosely detailed. Only selected key features of it are useful as human readable documentation.

Publications

1. Rich, C., Shrobe, Howard E., and Waters, Richard C. "An overview of the Programmer's Apprentice," Proceedings IJCAI-79, Tokyo, August 1979.

2. Shrobe, Howard E. "Dependency directed reasoning in the analysis of programs which modify complex data structures," Proceedings IJCAI-79, Tokyo, August 1979.

3. Waters, Richard C. "A method for automatically analyzing programs," Proceedings IJCAI-79, Tokyo, August 1979.

4. Waters, Richard C. "Programmer's Apprentice," in "Automatic Programming," Stanford Heuristic Programming Project Memo HPP-79-24, August 1979, Elschlager, R. and Phillips, J., eds., which is a section of the Handbook of Artificial Intelligence (Barr, A. and Feigenbaum, E., eds., to appear).

Theses in Progress

1. Rich, C. "Inspection methods in programming," Ph.D. dissertation, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1980.

2. Faust, Gregory G. "Semiautomatic translation of COBOL into HIBOL," S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected August 1980.

PROGRAMMING METHODOLOGY

Academic Staff

B. H. Liskov, Group Leader

Research Staff

P. R. Johnson

R. W. Scheifler

Graduate Students

R. R. Atkinson
V. A. Berzins
T. Bloom
M. P. Herlihy
T. O. Humphries
C. Johansson
D. Kapur
J. E. B. Moss
J. C. Schaffert
M. K. Srivas
E. W. Stark
W. E. Weihl

Undergraduate Students

M. D. Allen
G. H. Cooper
D. Ehrenfried
B. E. O'Connor

Support Staff

A. Rubin

Visitor

J. E. Stoy

PROGRAMMING METHODOLOGY

1. INTRODUCTION

This year work in the Programming Methodology Group has been focused on the design of a programming language to support the construction and execution of distributed programs. We have continued our investigation of the guardian model of distributed computation, and have concentrated on inter-guardian communication primitives, and on support for programs that continue to behave properly in the presence of node, network, and media crashes.

Our work in distributed computing is discussed in the following four sections. In Section 2, we discuss our goals and some underlying assumptions. In Section 3, we describe guardians, the basic modular unit that we provide for constructing distributed programs. Section 4 discusses some issues in message communication, and presents a method for sending the values of abstract objects in messages.

Section 5 describes the focus of our current research; it discusses problems that arise in trying to write robust programs, and describes some solutions to these problems. In Section 5, we identify the remote procedure call as a useful communication primitive, and define its semantic properties with respect to reliability.

In addition to our work in distributed computing, we have continued our study of program specification and verification. D. Kapur has completed research on providing a formal basis for specification and verification of programs that use and implement data abstractions; this work is discussed in Section 6. Section 7 discusses a system, constructed by R. Atkinson, that verifies synchronization properties of concurrent programs implemented using serializers. G. Stark has studied definitions of semaphores and the question of whether fair semaphores can be implemented using unfair semaphores; this work is discussed in Section 8.

Besides the work described here, we have also made considerable progress in our CLU implementation effort. A completely new implementation for the DECsystem-20 is now nearly finished. Production programs appear to be nearly twice as small and twice as fast, compared with our previous implementation. A major goal of the new implementation is to make CLU easily transportable to different machines and operating systems. We believe we have been fairly successful in meeting this goal. An initial implementation for the Motorola MC68000 is now ready for debugging. This implementation will serve as a base for our distributed CLU implementation.

2. GOALS AND ASSUMPTIONS

Technological advances have now made it possible to construct systems from collections of computers that communicate over a network. These advances call for the organization of software as *distributed programs*, whose modules reside and execute at several communicating yet geographically distinct locations. Our goal is to provide an integrated programming language and system to support the construction and execution of distributed programs.

Our approach is to extend an existing sequential language with primitives that support distributed programs. Our base language is CLU [1], [2]. CLU has been chosen for two reasons: it supports the construction of well-structured programs through its abstraction mechanisms, especially data abstractions, and it is an object-oriented language, in which programs are thought of as operating on long-lived objects, such as data bases and files -- a view well-suited to the applications of interest.

We have concentrated on applications that are concerned with manipulation of on-line data, e.g., airline reservation systems and banking systems, because we believe such applications are well-suited to a distributed implementation. Our intent is to design primitives that are as high level as possible yet are application-independent (within the chosen domain).

In the applications of interest, a major concern is to provide robust behavior in the face of failures of nodes, networks, and storage media. Accordingly, we have been studying linguistic support for robust software. This is a difficult problem and one that has been largely ignored in linguistic work, including current proposals for distributed programming constructs, e.g., [3], [4], [5], [6]. One exception is the work at Newcastle upon Tyne [7], [8], but that approach is concerned with recovery from user errors rather than failures. An *error* occurs when a software module does not meet its specification, while a *failure* is an expected, if not very desirable, outcome. Clearly it is possible to plan in advance how to deal with expected failure outcomes, but much more difficult (if not impossible) to cope with errors. We do not address the problem of error recovery.

In addition to our concern with reliability, we have a number of goals that are derived from the factors that make a distributed organization attractive in the applications of interest. Our approach must support extensibility in a natural way, so that, for example, the addition of new computers to the network can be handled gracefully. The programmer must be free to distribute processing and data in a way that satisfies availability and efficiency requirements. Also, the programmer must be able to protect sensitive data from misuse.

Our work is based on some assumptions about the underlying hardware. We assume that the distributed programs of interest run on *nodes* that are connected by means of a communications network. Each node consists of one or more processors, and one or more levels of memory. The nodes are heterogeneous, e.g., they may contain different processors, come in many different sizes and provide different capabilities, and be connected to different external devices.

The nodes communicate with each other only through the network; there is no (other) shared memory. We make no assumptions about the network. For example, it may be longhaul like the ARPANET [9] or shorthaul [10], or some combination with gateways in between; these distinctions are invisible at the programmer level.

We assume that each node has an owner with the authority to determine what that node does. For example, the owner may control what programs are allowed to run on that node. Furthermore, if the node provides a service to programs running on other nodes, that service may be available only at certain times, e.g., when the node is not busy running internal programs, and only to certain users. We refer to such nodes as *autonomous*.

The assumption of autonomy supports some of the goals mentioned in the Introduction. A consequence of this assumption is that the programmer, and not the system, must control where programs and data reside. Therefore, the language cannot hide completely the location of programs and data from the programmer. Furthermore, the system may not breach the autonomy of a node by moving processing to it for its own purposes. Our support for autonomy distinguishes our approach from multi-processor organizations such as CM* [11], and from high level appproaches such as the Actor system [12] where the mapping of a program to physical locations is entirely under system control.

3. THE GUARDIAN MODEL

To support the components of distributed programs, a modular unit is needed that can model the tasks and subtasks being performed in a natural way, and that can be realized efficiently. Toward that end we provide a construct called a *guardian*.

The purpose of a guardian is to provide controlled access to a resource or set of resources, e.g., by synchronizing concurrent accesses, and by checking access requests to determine if they should be allowed. A guardian contains *processes* and *data objects*. A process is the execution of a sequential program. The processes do the actual work of the guardian. They manipulate the data objects, and can communicate with one another via shared objects.

Many guardians may cooperate to provide a *subsystem*: an application or system service such as a distributed data base or a message system (see [13] for an example). However, processes in different guardians can communicate only by sending *messages*. Messages contain the *values* of objects, e.g., "2" or " # 176538 $173.72" (the value of a bank account object). An important restriction ensures that the address space of a guardian remains local: it is impossible to place the *address* of an object in a message. It is possible to send a *token* for an object in a message. A token is an external name for the object, which can be returned to the guardian that owns the object to request some manipulation of that object. (A token is a sealed capability [14] that can be unsealed only by the creating guardian.) The system makes no guarantee that the object named by the token continues to exist; only the guardian can provide such a guarantee. Thus a guardian is entirely in charge of its address space, and the system can perform storage management locally for each guardian.

Although a subsystem may make use of guardians at many nodes, each individual guardian exists entirely at a single node of the underlying distributed system: its objects are all stored on the memory devices of this node and its processes run on the processors of the node. During the course of a computation, the population of guardians will vary; new guardians will be created, and existing guardians may go away. Since the sizes of the physical nodes may vary, different nodes will support different numbers of guardians. A guardian may be created at a node only by (a process in) a guardian at that node. Each node comes into existence with a *primal* guardian, which can, if the owner of the node wishes, create guardians at its node in response to messages arriving from guardians at other nodes. This restriction on creation of new guardians helps preserve the autonomy of the physical nodes. Guardians may move from one node to another, but in a similarly restricted way.

A guardian is an abstraction of a physical node of the underlying network: it supports one or more processes (abstract processors) sharing private memory, and communicates with other guardians (abstract nodes) only by sending messages. In thinking about a distributed program, a programmer can conceive of it as a set of abstract nodes, each of which performs a meaningful task for its application. Intra-guardian activity is local and inexpensive (since it all takes place at a single physical node); inter-guardian processing is likely to be more costly, but the possibility of this added expense is evident in the program structure. The programmer can control the placement of data and programs by creating guardians at appropriate nodes. Furthermore, each guardian acts as an autonomous unit, guarding its resource and responding to requests as it sees fit.

4. COMMUNICATION

We have found it useful to distinguish two issues within communication. The first concerns the form of messages, and addresses such problems as type checking of message communication and the kinds of values that can be sent in messages. The second concerns important semantic properties of communication, including such questions as order of arrival of messages, and, most importantly, the reliability of communication. In this section we discuss the first issue; discussion of the second issue is deferred to Section 5.2.

We believe that message communication should share certain desirable properties with the conventional procedure call, namely the ability to do compile-time type checking of message communication, and the ability for programs to communicate in application-oriented terms (i.e., in terms of the abstract objects of interest in the application program). The properties that the communication primitive should provide are as follows.

1) User programs need not deal with the underlying form of messages. For example, users should not need to translate data into bit strings suitable for transmission, or to break up messages into packets.

2) All messages received by user programs are intact and in good condition. For example, if messages are broken into packets, then the system only delivers a message if all packets arrived at the receiving node, and were properly reassembled. Furthermore, if the bits in a message have been scrambled, the message is either not delivered, or reconstructed before delivery; clearly some redundant information is required here.

3) Messages received by a module are the kind that module expects. Support for this property requires type checking, which may be performed either at compile time or run time. Performing such type checking is analogous to type checking procedure calls.

4) Modules are not restricted to communicating only in terms of a pre-defined set of types, e.g., the built-in ones. Instead, modules can communicate in terms of values of interest to the application. In particular, if the application is defined using abstract data types, then values of these types can be communicated in messages.

Type checking at compile time can be supported by a declarative mechanism similar to a header of a procedure definition; the compiler can use this to check that

the form of a message satisfies the type constraints imposed by the intended recipient. In our language, messages are sent to ports. Each port has a type that completely determines the set of messages it can receive and the responses to those messages. A guardian definition lists one or more port types to be provided for communication. This information is sufficient to permit compile-time type checking of message passing. Compile-time checking is possible even if guardian definitions are compiled separately, provided that compilation is done in the context of a library containing descriptions of guardians. CLU already is based on such a library.

The above four properties are needed to give message communication the same status as procedure call. The properties are supported to some extent in all languages that provide linguistic primitives for message communication. For example, ADA [5] and requires all four properties. However, none of these languages has addressed the question of how to communicate the abstract values. We discuss such a method below. To simplify the discussion, we ignore the problems caused by shared and cyclic data structures. A description of the full method can be found in [15], [16].

4.1. Communicating Abstract Values

An abstract data type consists of a set of abstract objects and a set of primitive operations to manipulate those objects. The implementation of an abstract data type must define an internal representation for the objects, and then define implementations for the operations in terms of the chosen representation. One possibility for communicating abstract values (the values of the abstract objects) is to communicate their internal representations. Although such a representation may be defined in terms of other abstract data types, ultimately it is expressed in terms of the built-in types of the language. Thus the representation can be transmitted assuming (as we do) that the built-in types can be transmitted.

Nevertheless, transmitting abstract values by transmitting their representations is unsatisfactory for a number of reasons:

1) Different guardians may use different representations for values of user-defined types. Security concerns, differing patterns of use, and differing hardware characteristics may all encourage guardians to "customize" their implementations of abstract types, while retaining the need to exchange such values with each other. The direct transmission of underlying representations clearly precludes the possibility of constructing customized representations for abstract types.

2) The underlying representation of a value may be transmissible, while the value itself may not be. For example, a value of type "filename" may be represented by a character string. The string may be transmissible, but the filename may be meaningless outside of a particular file system residing at a particular node.

3) Conversely, a value may be transmissible, while its representation may be unsuited as a vehicle for communication. For example, a value's representation may contain information meaningless to another guardian, such as an index into a private table belonging to the original guardian. Or, a value's representation could include values that are not themselves transmissible but that can be reconstructed by the recipient.

We conclude that what is needed is user control over transmission of abstract values. In the remainder of this section, we discuss a method that supports user control. This method satisfies the following design goals:

Modularity. Transmission of the values of a data abstraction is defined locally as part of the implementation of that data abstraction. This principle is necessary to ensure that knowledge of the representation is local to the implementation.

Useability. The definition of transmission can be given in terms of any convenient data types. The programmer does not need to translate into strings of bits.

Linearity. The work needed to implement value transmission is linear in the number of implementations for the data abstraction.

The transmission method is described in terms of CLU, but is applicable to any language that supports data abstraction.

For purposes of discussion, let us assume that messages are sent by executing

send $C(a_1, ..., a_n)$ **to** p

Here p is a port attached to some guardian G, C is the name of a request or operation that G is being asked to perform, and $a_1, ..., a_n$ are the actual arguments of operation C. The effect of executing such a **send** statement is that a message containing the name C and the values of $a_1, ..., a_n$ is constructed and sent to p. To acquire an incoming message, G executes a receive statement:

```
receive on p
    when C (f1: T1, ..., fn: Tn): S
    ...
    end
```

When a message containing the request named *C* is received, the associated statement S is executed, with the formals f_1, ..., f_n initialized to contain the values extracted from the message. (Properties (2) and (3) discussed above ensure that requests named *C*, with arguments of types T_i, can be sent to *p* and received from *p*.)

Our method for value transmission is straightforward. For every abstract type that is *transmissible*, we require that an *external representation* be defined. The external representation is just another type, either built-in or abstract. The meaning of the external representation is that values of this type are to be used in representing the abstract values in messages.

The external representation is distinct from the internal representation used in implementations of the abstract type, but these implementations are written with knowledge of the external representation and the relationship of its values to the abstract values. Each implementation must provide two operations to map between (its internal representation of) values of abstract type, *T*, and external representation type, *XT*. There is an operation

encode = **proc** (t: T) **returns** (XT)

to map from *T* values to *XT* values (for sending messages) and an operation

decode = **proc** (x: XT) **returns** (T)

to map from *XT* values to *T* values (for receiving messages). *Encode* and *decode* are correct if they preserve the abstract values: *encode* must map (an internal representation of) a value *v* of type *T* into an *XT* value that represents *v*, while *decode* must perform the reverse mapping. (These correctness criteria are discussed in detail in [16].)

Encode and *decode* are not called explicitly by user programs; instead, the language implementation makes these calls in the course of sending and receiving messages. Each actual argument in a **send** statement must be of a transmissible type. If this type is one of the built-in types, then the system knows how to place its value in a message and how to extract its value from a message. For abstract types, the system calls *encode* repeatedly until the argument has been translated to a value of built-in type. This value is then transmitted in the message. When a message is received, the reverse happens: a value of built-in type is extracted from the message, and then *decode* is called repeatedly until a *T* value is obtained. (The language implementation is discussed in [15].)

As an example of a typical user-defined type, we consider complex numbers. This type provides operations to create new complex numbers, to add, subtract, multiply and divide complex numbers, to compare complex numbers, and to obtain the real and imaginary coordinates of a complex number. Both rectangular and polar coordinates are useful representations for complex numbers. The choice of representation depends on the relative frequency of addition vs. multiplication.

A good external representation for complex numbers might allow either coordinate system to be used. In CLU, this would be expressed by giving type definitions, e.g.,

```
xrep = oneof [xy: xycoords, polar: polarcoords]
xycoords = record [x,y: real]
polarcoords = record [rho, theta: real]
```

Here **xrep** is a reserved word that identifies the external representation type. A *oneof* is a built-in CLU type similar to a variant record; an object of this type can be either of type *xycoords*, in which case it is tagged by the identifier *xy*, or of type *polarcoords*, tagged by the identifier *polar*.

Figure 1 shows part of a CLU cluster that implements complex numbers using rectangular coordinates as the internal representation (**rep**). Here *encode* obtains the internal representation of a complex number (via **cvt**) and builds the **xrep** for this number, using the *xy* variant. *Decode* must check the variant of the **xrep** value it receives, and do a conversion to rectangular coordinates if it receives the polar form. The internal representation it constructs turns into a complex number when it is returned (via **cvt**).[1] The implementation of *encode* is obviously correct, and so is the implementation of *decode* when an *xy* variant is received. To show the correctness of *decode* in the other case requires a proof that the *xy* pair constructed by *decode* represents a complex number sufficiently close to the complex number represented by the *polar* pair input to *decode*.

5. RELIABILITY

In the applications of interest, important information is entrusted to the system. It is crucial that such information not be lost if various failures occur. In this section, we explore the issues that arise in trying to write programs that are robust--survive

[1] **Cvt** indicates a conversion between abstract and internal representation type. **Cvt** may only appear in a cluster, and maps between the **rep** of that cluster, and the abstract type implemented by that cluster. When **cvt** appears as the type of an argument, the mapping is from abstract type to **rep** type; when it appears as a result type, the mapping is the other way. See [1], [2] for more information.

```
xrep = oneof [xy: xycoords, polar: polarcoords]
xycoords = record [x,y: real]
polarcoords = record [rho, theta: real]

complex = cluster is create, real, imag, add, sub, mul, divide, equal, ...

    rep = xycoords

    encode = proc (c: cvt) returns (xrep)
        return (xrep$make – xy (c))
        end

    decode = proc (xc: xrep) returns (cvt)
        tagcase xc
            when xy (p: xycoords): return (p)
            when polar (p: polarcoords):  r: real := p.rho * cos (p.theta)
                                          i: real := p.rho * sin (p.theta)
                                          return (rep$ {x:r, y:i})
            end
        end decode

    % definitions of procedures implementing the operations listed in
    % the header appear here

end complex
```

Figure 1: Complex number example

node, network and storage media failures without loss of essential information. We also outline our approach to supporting the writing of robust software.

Our assumptions about node, network and media failures are the same as those discussed in [17]. Roughly, we assume that failures are detectable. This assumption is reasonable for network and media failures, but less so for node failures: we cannot cope with a node that runs erroneously instead of crashing.

5.1. Permanence of Effect

In a distributed program, modules at different nodes may interact to achieve some common end. In our model, since guardians have no common memory, this interaction is achieved by message passing. One guardian sends a message to another requesting it to perform some action. If, subsequently, the requesting

guardian is notified (by some other message) of the effect of the action, it must be able to rely on this information. Partly, the concern is one of program correctness: both requester and server must agree about the meaning of the message exchange. However, there is an additional issue here concerning reliability: it is important that the reported effect not be undone by subsequent (node, media, or possibly network) failures. We refer to the desired property as *permanence of effect*. Permanence is needed in both centralized and distributed programs, but the need seems to be particularly acute in distributed programs, because different parts of such programs can fail independently.

Our approach to permanence of effect is to provide guardians with a means of ensuring that their data survives crashes. Each guardian definition can declare a set of permanent variables. The guardian's permanent state consists of these variables, and all the data reachable[2] from them. The permanent state is stored in primary memory, and the data in it may be modified in the ordinary way. However, a backup copy of this information resides in non-volatile memory,[3] and the guardian has the ability to control when this copy is changed. For purposes of discussion, we will assume the existence of a **save** primitive. To change the copy, the guardian executes the **save** primitive. When this primitive is called, it stores an image of the permanent state in non-volatile storage. The store is done *atomically*: either the entire permanent state is saved, or the effect is as if the save had not been started. Thus a crash in the middle of a save does not leave the backup copy of the guardian's permanent state inconsistent.

Note that we are not talking here about a virtual memory scheme, where the system is moving pages between the secondary and primary memory. Such a scheme cannot ensure that the copy on secondary memory is in a consistent state at the time of a crash. Instead, a primitive like **save** is essential.

The system guarantees that guardians themselves are permanent. This means that after a node crash, the system will cause all guardians running at that node before the crash to continue their existence. Each guardian will restart with its permanent state having the value it possessed at the last completed save before the crash.

A guardian definition has two code sections. The first is the **init** section; its purpose is to initialize the permanent state to some consistent, initial value. The **init**

---

[2] The notion of reachability arises because there may be pointers.

[3] This non-volatile memory could be located at another physical node.

section runs whenever a new guardian is created. Only after **init** is finished is the guardian in a fit condition for surviving crashes; at this point, the guardian becomes permanent.

The second section is the **start** section. This section runs when **init** is complete, and also whenever the guardian is restarted after a crash. Its function is first to initialize whatever volatile state the guardian uses (for example, the guardian may keep an inverted index to a data base for fast retrieval), and then to do its actual work, e.g., continue what it was doing before the crash, and respond to incoming request messages.

An example of a simple guardian definition is given in Figure 2 to illustrate these concepts. The syntax here is not intended to be real, but is introduced just for the example. This guardian provides an unbounded buffer of items. The buffer is stored in an array, which constitutes the permanent state. The volatile state is simply a count of the current number of elements; this information is redundant with information in the array. The communication primitive used in this example pairs requests and responses; when the **reply** statement is executed, the system sends the reply message to the process that made the request.

Although very simple, this example illustrates a common property of guardians, namely that in response to requests, guardians perform mapping from a consistent permanent state to a new, consistent, permanent state. The example also illustrates that the mechanism used in the simple way shown is not powerful enough to prevent obvious problems. For example, a node crash between the time the code handling *get* performs **save** and the time it replies will cause the item to be lost. Such problems will be discussed in the next section.

It is worth noting that the above mechanism is tied to guardians and not to processes. The system creates a single process inside the guardian to run the **init** or **start** code; however, this process can fork others, so that many processes can be running concurrently inside a guardian. (These processes must synchronize with each other as needed.) Nevertheless, when one of these processes executes **save**, what is saved is the permanent guardian state, and not the state of the process. Saving just guardian state is a convenient way of giving the programmer close control over the amount of permanent state. One possible negative result of this decision is that sometimes information about processes must be encoded in the permanent state, so the processes can be restarted (by the **start** code) after a crash. In the examples we have studied so far, it has been natural and easy to save information about processes; the programmer need not save the equivalent of a process checkpoint, but instead records tasks that the guardian is working on.

```
buffer = guardian
    buf = array [item]
    pport = port [put (item) replies (ok)]
    gport = port [get ( ) replies (ok(item))]
    permanent
        b: buf
        p: pport
        g: gport
    init ( ) replies (ok (pport, gport))
        b := buf$new ( )
        p := pport$create ( )
        g := gport$create ( )
        save
        reply ok (p, g)
    start
        count: int := buf$size (b)
        while true do
            if count = 0 then
                receive on p
                    when put (i: item):   % add item to end of buffer.
                                          buf$addh (b, i)
                                          save
                                          count := count + 1
                                          reply ok
                    end
                end % if
            receive on p, g
                when put (i: item): buf$addh (b, i)
                                    save
                                    count := count + 1
                                    reply ok
                when get ( ):       % remove and return first item from buffer
                                    i: item := buf$reml (b)
                                    save
                                    count := count - 1
                                    reply ok (i)
                end
            end % while
    end buffer
```

Figure 2: A guardian that provides a buffer

A few remarks are in order about the "non-volatile" storage used to store permanent data. We believe that the reliability of this storage may vary from node to node. The information could be stored in such a way that it survives node failures, or failures of the node and a single media device, or failures of two media devices, etc. Whatever the storage method, reliability will never be 100%. To obtain reliability high enough for the needs of an application, nodes with the desired reliability properties could be purchased. Alternatively, critical data could be duplicated by storing it as the permanent state of guardians at other nodes. For example, in a data base system, a duplicate copy of the log might be kept at another guardian.

5.2. Inter-Guardian Consistency

As was mentioned above, the permanence mechanism does not solve all problems. The basic difficulty is that the mechanism allows a single guardian to make transitions from one permanent, consistent state to another, while what is needed is a mechanism that allows groups of guardians to make such transitions. For example, the user of a buffer guardian requests an item, and the buffer guardian provides one. The user then uses this item, and finally makes a change in its permanent storage to record the result. Only at this point should the item be truly removed from buffer guardian's permanent storage, because only at this point are the two guardians in a mutually consistent state: the item has truly been consumed.

We have been studying the problem of support for inter-guardian consistency. There are a number of difficulties that arise, some of which are discussed below. In the following, *B* is a buffer guardian and *U* is its user.

1) Suppose *B*'s node crashes before *U* gets a reply. Does the system hide this from *U*, or must *U* send the request again (after a suitable time has elapsed)?

2) If either *U* or the system sends the request again, there is a possibility that *B* already acted on the previous request. Must *B* recognize duplicate requests, or does the system hide this by never presenting *B* with a duplicate?

3) If *B* crashes after a **save** but before a reply to the *get* message, what prevents the item from being lost? Must *B* prevent this, or does the system prevent it?

4) If *U*'s node crashes after the request is sent, what ensures that after the crash, *U* picks up from where it left off in interacting with *B*, assuming *U* is still interested in the request?

5) In fact, *U* may not be interested in the request if the answer takes too long in coming, either because of a crash (at either node), or just because the person *U* was working for got tired of waiting. In this case, how is work performed for *U* by *B* undone, i.e., the effect of the request(s) removed from *B*'s permanent storage?

Note that in all these cases, the questions concern whether the system or the programmer handles the problems. Furthermore, all the questions are phrased in terms of communication between *U* and *B*. So what is at issue here is the exact semantics of message communication, and the relationship of message communication to permanent storage.

We return now to the issue, raised in Section 4, of the reliability of message communication. We believe one viable approach to this issue is to provide a communication primitive with no additional reliability properties over what the underlying network provides. The primitive might pair requests with responses, or it might not. In either case, the primitive would provide the properties discussed in Section 4, but it would *not* hide the unreliability of the network or the nodes. It would not hide the fact that messages may be duplicated or arrive out of order. Most importantly, it would not guarantee message delivery.

Such a level is reasonable because it incurs no additional expense over what is necessary for communication in application-oriented terms. However, it provides no help to the programmer of either *U* or *B* in solving the problems discussed above.

We have studied how various applications might cope with the above problems while using such a primitive in conjunction with the permanence mechanism of Section 5.1. For example, *U* might periodically re-send the request message, while *B* would check for duplicates. Sometimes duplicates are not a problem; this happens for requests that are naturally *idempotent* (many executions are equivalent to one) [17]. For non-idempotent requests, such as *put* and *get* for the buffer, the request must have as an extra argument a unique identifier that can be used to recognize duplicates. *B* can use the unique identifier to remember its previous response to the request, and send it again if the request comes in again. The information about requests and responses must be stored in permanent storage; however, saving it before the reply, and then changing the array and saving the change after the reply, solves problem (3). To avoid having to remember old requests forever, *U* and *B* may have to resort to a protocol that allows *U* to inform *B* that old requests can be forgotten.

So far, it appears that the programmer only encounters awkwardness, but no real difficulty, in solving the above problems (although we might be a little concerned that

there are too many **saves** going on). However, this appearance is deceptive, because we have not yet solved problems (4) and (5), and they are the difficult ones. Furthermore, we have been looking at a limited kind of interaction, involving just two guardians, a *client* requesting service, and a *server* providing service. In general, we must expect nesting to occur, e.g., the client may actually be performing a service for some higher level client. *U* might be such an intermediate server/client.

Consider the case where *U*'s node crashes after *B* has performed a request but before *U* has recorded the reply in its permanent storage. When, after the crash, *U* re-receives the message that caused it to send the request to *B*, it must be able to re-send the request to *B* with sufficient information so that *B* can identify that request as a duplicate. Another possibility is that *U*'s client may not re-send the request to *U* or may ask *U* to abandon the request (problem 5 above). In this case, *U* (or some other guardian) must send a message to *B* requesting it to undo the previous work.

The analysis above is actually oversimplified, because we have not considered synchronization requirements. For example, often a read operation is not really idempotent, since usually it is important whether the read is done before or after a write operation on the same data. Also, if *U* adds an item to *B*'s buffer, this item should not be available to any other of *B*'s clients until *U* (or *U*'s client) is really finished.

With the addition of some sort of synchronization method, programmers can solve all the above problems. However, to do so requires substantial work, including both bookkeeping and an agreement about protocols between all cooperating guardians. Roughly, the guardians must carry out a two-phase commit protocol, complete with intentions lists [17] or undo/redo logs [18]. To perform such a protocol correctly requires careful analysis, and shortcuts usually result in errors, while to perform it efficiently is a difficult systems problem requiring substantial ingenuity. Therefore, it seems appropriate to attempt to provide primitives that make this work part of the language implementation, and hide it from the programmer.

We intend to support higher level primitives that provide the desired semantics. One of these primitives is a *remote procedure call* (RPC). RPC pairs requests and replies, and re-sends messages as needed. More importantly, however, RPC provides *at-most-once* semantics: If the caller receives a reply, the system guarantees that the call was acted on exactly once, without any programmer having to worry about providing this. For example, a programmer need not worry about recognizing duplicates, because the system won't deliver any. Furthermore, if the call was not completed, either because the caller lost interest, or because it was not possible to do the requested work, the RPC is automatically undone, and guaranteed to have no effect.

It is worth noting that at-most-once semantics are just what we require of an ordinary procedure call in the presence of node crashes, permanent data, and impatient users. Furthermore, all the problems discussed above arise in a centralized system. In the past, however, the semantics of procedure call have not taken these factors into account.

RPC is a very high level primitive, certainly much higher level than the primitive discussed above, which provided only the reliability properties of the underlying network. For some time we were hopeful that there might be an intermediate level primitive that would solve many of the user's problems, and would not be as expensive as RPC. Our experience indicates that there is no such primitive; we have looked for one and have not found it.

As a simple example, consider a primitive that doesn't guarantee message delivery, but does guarantee that messages arrive in order, and that duplicate messages created by the network are not delivered to the user. Such a primitive is easy and fairly cheap to implement, but not very helpful to the user because the user must be concerned with detecting duplicates at a higher level. There will be duplicates generated by the user in attempting to guarantee delivery. For example, *U* was generating such duplicates in the scenario above. And, there will be duplicates created by the user after a crash, when a previous request is re-tried. Since the user must worry about duplicates anyway, the logic of his program is not simplified by the work the system is doing. It is true that the order preserving property could be used to control how long old requests must be remembered; however, the unique identifiers mentioned above must still be put in the messages and can also be used for this purpose. So although in this case it was not expensive to implement the communication primitive, the work being done is wasted effort.

In fact, intermediate level primitives often seem to be both of little help to the user, and expensive to implement. Therefore, we believe that only the two extremes are reasonable choices.

6. A THEORY FOR ABSTRACT DATA TYPES

Kapur [19] has developed a rigorous theoretical framework for studying various aspects of immutable abstract data types. The main contribution of this research is a framework for abstract data types that is rigorous and that brings together various aspects of abstract data types in a unified and coherent way. The framework incorporates important and useful features such as hierarchical structure and modularity. It is also broader in scope than other similar work, in that it handles data types with nondeterministic operations and with operations exhibiting exceptional

behavior. The framework will be useful to a designer of a specification language for abstract data types as it provides a semantic basis for studying and comparing such specification languages. It also provides a formal basis of automatic deductive systems for abstract data types. Furthermore, this research clarifies our intuititions about data type behavior and provides a rigorous foundation for them; as examples, notions of consistency and completeness of a specification, and the correctness criterion for an implementation, can be stated formally and analyzed.

The central notion of this framework is the *definition* of an abstract data type as an equivalence class of heterogenous algebras. The notion of a *specification* is distinct from that of a data type; in fact, a single specification may be satisfied by several different data types. The framework also provides a *deductive system* for deriving properties of an abstract data type from its specification, and for proving the correctness of an implementation. The separation of the definition of an abstract data type from development of a specification language and deductive system is crucial. Such a separation makes it meaningful to consider notions of consistency and completeness of a specification, and to determine the "power" of the deductive system.

In the sections below, we give an informal description of the important components of the framework. The description is restricted to data types whose operations do not signal any exceptions; for a detailed study of exceptions see [19]. The next section discusses the definition of a data type, and gives an algebra modeling the data type **IntSet** of integer sets. The second section presents the specification language by giving a formal specification of **IntSet**. The final section discusses the deductive system, explaining how a theory is constructed for **IntSet**. For discussions of the various notions of completeness and consistency of specifications, and for a definition of the correctness criterion for an implementation, refer to [19].

6.1. Definition of a Data Type

The central notion in the framework is the *definition* of a data type. The definition of a data type is distinct from its specification. We use heterogeneous algebras (*type algebras*) to define a data type, since they can be used to model a data type naturally and elegantly. The definition method is hierarchical. Usually a data type is defined in terms of a number of other existing types. These types are called the *defining types* for the new type. The terminal point in this hierarchy is the boolean type, which does not have any defining types.

Figure 3 shows a type algebra A_{si} modeling the data type **IntSet**. This algebra

uses the set of finite sequences of nonrepeating integers to represent elements of the data type, and the sets **Z** and **B** to represent the values of the defining types integer and boolean. The function **Choose**, which chooses an arbitrary element from a set, is nondeterministic.

A_{si} = [{ **SQ**, **Z**, **B** }; {Null, Insert, Remove, Has, Size, Choose }],

where **SQ** = { ⟨⟩, ⟨0⟩, ⟨1⟩, ⟨-1⟩, ⟨2⟩, ⟨-2⟩, ⟨0, 1⟩, ⟨0, -1⟩,
⟨0, 2⟩, ⟨0, -2⟩, ⟨1, 0⟩, ⟨1, -1⟩, ⟨1, 2⟩, ⟨1, -2⟩, ⟨-1, 0⟩, ⟨-1, 1⟩,
⟨-1, 2⟩, ⟨-1, -2⟩, ⟨2, 0⟩, ⟨2, 1⟩, ...},
Z = { 0, 1, -1, 2, -2, ...},
B = { true, false }

Let $s = \langle i_1, ..., i_m \rangle$, $m \geq 0$; if $m = 0$, then $s = \langle\rangle$.

Null	$\triangleq$	⟨⟩	
Insert($\langle i_1, ..., i_m \rangle$, i)	$\triangleq$	$\langle i_1, ..., i_m \rangle$	$\exists\, 1 \leq j \leq m, i_j = i$
	$\triangleq$	$\langle i_1, ..., i_m, i \rangle$	otherwise
Remove($\langle i_1, ..., i_m \rangle$, i)	$\triangleq$	$\langle i_1, ..., i_{j-1}, i_{j+1}, ..., i_m \rangle$	$\exists\, 1 \leq j \leq m, i_j = i$
	$\triangleq$	$\langle i_1, ..., i_m \rangle$	otherwise
Has(s, i)	$\triangleq$	true	$\exists\, 1 \leq j \leq m, i_j = i$
	$\triangleq$	false	otherwise
Size(s)	$\triangleq$	m	
Choose($\langle i_1, ..., i_m \rangle$)	$\triangleq$	i_j	$1 \leq j \leq m > 0$

Figure 3: A Model of **IntSet**

Note that an element of **IntSet** may have multiple representations in this model. For example, both ⟨1, 2⟩ and ⟨2, 1⟩ represent the set **{1, 2}**.

A data type can be modeled using several different type algebras, each using different representational structures. However, we are interested only in the *behavioral view* of a data type, which is the behavior detected via the operations. This view abstracts away from different representational structures, as well as from multiple representations of value. The definition of a data type is interpreted so as to reflect this view.

First we define *observable equivalence* on values: two values are observably equivalent if they cannot be distinguished using operations of the type. For example, the values ⟨1, 2⟩ and ⟨2, 1⟩ in the type algebra for **IntSet** are observably equivalent. To abstract from multiple representations of values, we define a reduced algebra by taking the quotient of each of the domains in the type algebra with respect to the observable equivalence relation. We abstract from the representational structure by

defining *behavioral equivalence* on type algebras: two type algebras are behaviorally equivalent if their reduced algebras are isomorphic. Informally speaking, two type algebras are behaviorally equivalent if both can be used to simulate the behavioral view of the same data type. A data type is then defined as a set of behaviorally equivalent type algebras.

6.2. Specification Language

Our framework uses an axiomatic method to specify data types. Figure 4 shows a specification of **IntSet**. The *operations* component specifies syntactic properties of the operations, such as the domain and range of each operation, abbreviations (if any) for operation names, and indicates which operations are nondeterministic.

Operations

Null	: --> IntSet	*as* ø
Insert	: IntSet X Int --> IntSet	
Remove	: IntSet X Int --> IntSet	
Has	: IntSet X Int --> Bool	*as* $x_2 \in x_1$
Size	: IntSet --> Int	as # (x_1)
Choose	: IntSet --> Int	*nondeterministic*

Restrictions

Pre (Choose(s)): # (s) > 0

Axioms

Remove(ø, i) ≡ ø
Remove(Insert(s, i1), i2) ≡ **if** i1 = i2
then Remove(s, i1) **else** Insert(Remove(s, i2), i1)
i ∈ ø ≡ F
i1 ∈ Insert(s, i2) ≡ **if** i1 = i2 **then** T **else** i1 ∈ s
(ø) ≡ 0
(Insert(s, i)) ≡ **if** i ∈ s **then** # (s) **else** # (s) + 1
Choose(s) ∈ s ≡ T

Figure 4: Specification of **IntSet**

The *restrictions* component specifies restrictions on operations. For example, in Fig. 4 this component specifies as a precondition for the **Choose** operation that its argument must be nonempty. This means that an axiom involving **Choose** in the *axioms* component holds only when the argument to **Choose** is nonempty.

The *axioms* component specifies the semantic properties that the operations satisfy. It consists of a set of equations (or axioms) of the form '$e_1 \equiv e_2$' where e_1 and e_2 are expressions of the same type, and e_2 may be a conditional. Informally '$e_1 \equiv e_2$' means that for all instantiations (satisfying the *restrictions*) of the variables in e_1 and e_2, the expressions interpret to observably equivalent values. When the expressions involve nondeterministic operations, the expressions can interpret to more than one value. In such a case the axioms have the following interpretation: for every possible interpretation of e_1, there exists an observably equivalent interpretation of e_2, and vice versa. Thus, the equations in effect attempt to capture the observable equivalence relation of the type being specified. In fact, the symbol '$\equiv$' denotes the observable equivalence relation on the appropriate domain.

The semantics of a properly designed specification is a set of related data types which differ in the behavior not captured by the specification. If an operation is specified to be nondeterministic, the semantics of a specification includes data types in which the operation can have as much nondeterminism as desired. For instance, the data types that satisfy the specification in Fig. 4 may differ in the behavior of the **Choose** operation on an empty set, and also in the amount of nondeterminism **Choose** exhibits.

6.3. Deductive System

The deductive system is based on first order multi-sorted predicate calculus with identity. It is used to derive properties of a data type from its specification and also in proving the correctness of an implementation. The components of the deductive system are derived from the specification.

The nonlogical axioms of the system are derived from the equations in the *axioms* part of the specification. If an equation involves neither any nondeterministic operations nor any operations with preconditions, the equation itself can serve as a nonlogical axiom. This is because the equation does not need any special interpretation. On the other hand, when an equation contains a nondeterministic operation symbol, the equation must be modified so that the resulting formula reflects the special semantics associated with the equation. This is done by using a predicate characterizing the relation expressed by the nondeterministic operation. For **IntSet** all but the last equation, involving **Choose**, can be used directly as nonlogical axioms. The nonlogical axiom corresponding to the last equation is:

$$(\forall\, i,\, [\sim\, \#(s) = 0 \equiv T \Rightarrow (\mathbf{ChooseP}(s, i) \equiv T \Rightarrow i \in s \equiv T)] \wedge$$
$$(\exists\, i)\, [\sim\, \#(s) = 0 \equiv T \Rightarrow \mathbf{ChooseP}(s, i) \equiv T]$$

where **ChooseP**(*s*, *i*) is true if *i* is a possible result of **Choose**(*s*). The nonlogical

inference rules of the deductive system characterize general properties of data types which are assumed implicitly in the specification. There is a set of rules capturing the equivalence relation property of the observable equivalence '$\equiv$'. The system has a method of proof by contradiction that can be used to prove inequalities of the form '$e_1 \not\equiv e_2$'. The system has an infinite induction rule capturing the property that every value of a data type is constructed by a finite application of the operations of the data type. The induction rule for **IntSet** can be described informally as follows. If a property **P** holds for all possible ground terms, i.e., expressions without variables, formed out of the operations **Null**, **Insert**, and **Remove**, then it can be inferred that **P** holds for all values of **IntSet**.

An important advantage of developing a formal deductive system for a data type is that it is possible to state precisely the completeness and consistency properties of a specification. A specification is consistent if for any two ground terms e_1 and e_2, both $e_1 \equiv e_2$ and $e_1 \not\equiv e_2$ cannot be proved in the theory. A specification is complete if for any two ground terms e_1 and e_2 of the same type, either $e_1 \equiv e_2$ or $e_1 \not\equiv e_2$ is derivable in the theory.

7. AUTOMATIC VERIFICATION OF SERIALIZERS

In systems where several processes may attempt to access the same resource concurrently, there is usually a need to impose some order on those accesses. If certain orders are not enforced, certain classes of access to the resource may conflict and cause erroneous results. However, other classes of access to the same resource may proceed concurrently without conflict.

Atkinson [20] has studied the use of *serializers* [21] in controlling concurrent access to shared resources, and has developed a program for automatically verifying a simple class of serializers. The task of such a verifier is to prove that programs obtain correct results when concurrently manipulating shared resources, under the assumption that individual accesses to a resource obtain the correct results in the absence of concurrency. In particular, the verification process is used to show that certain accesses exclude others, that proper accesses are granted priority, that appropriate accesses may proceed concurrently, that there is no deadlock, and that there is no starvation.

In the next two sections we present a restricted form of serializer and an abbreviated semantic model for the concurrent execution of programs. The third section describes the language used to specify the correct behavior of serializers, and gives a number of sample specifications. The fourth section briefly outlines the structure of the verifier.

7.1. Simple Serializers

The serializer construct is intended to provide a modular method of concurrent access to shared data objects. A serializer is similar to a cluster [1], in that it defines a new data type. The objects of such a data type are called *serializer objects*. Each serializer object is used to control a separate resource object. Operations are provided to create new serializer objects, and to access the resource controlled by an existing serializer object.

Certain parts of access operations are *protected*, in that execution of protected code by one process for a particular serializer object precludes the concurrent execution of protected code by any other process for the same serializer object. The process executing protected code is said to have *possession* of the serializer object.

When a process first starts executing a serializer access operation, it must gain possession of the serializer object. This is done by waiting on an implicit *external* queue, which is serviced in first-in-first-out order. During execution, possession of the serializer object can be released and regained. Possession is released while accessing the resource, thereby permitting concurrent activity involving the serializer object, and then regained by waiting on the external queue. It also may be necessary to suspend execution to wait for some condition to become true. For example, an operation needing exclusive access to the resource must wait until no other resource accesses are in progress. This is accomplished by waiting on an explicit queue, and releasing possession of the serializer object to allow other requests to proceed concurrently as far as they are able.

A *simple* serializer object consists of a fixed number of *crowds*, a fixed number of *queues*, and a resource object. Crowds are used to define classes of access on the resource object, and to record which processes are performing which class of access. Queues are used to impose a first-in-first-out discipline on processes waiting for conditions.

The body of a simple serializer access operation consists of a sequence of **enqueue** and **join** statements. An **enqueue** statement is used to wait until some condition holds. This statement has the form:

enqueue *queue – expression* **until** *condition*

The condition must be composed of the logical and (&) and the logical or (|) of the following two forms of expressions:

crowd$empty(*crowd – expression*)
queue$empty(*queue – expression*)

The executing process is always placed on the specified queue, even if the specified

condition currently holds. Possession of the serializer is then released. Whenever possession is released, all explicit serializer queues are examined to determine whether any queue is *ready*, i.e., has a process at its head with a true condition. If so, one of these processes is removed from its queue and gains possession. If not, a process is removed from the external queue and gains possession.

A **join** statement is used to release possession and access the resource. This statement has the form:

join *crowd – expression* **do** *resource – invocation* **end**

The **join** statement starts by placing some identification of the executing process into the specified crowd and releasing possession. The resource invocation is then performed, after which the process is placed on the external queue. When possession is regained, the process identification is removed from the crowd, and execution continues with the next statement after the **join** statement. The results of the resource access may be returned as the results of the serializer operation. This is indicated in the body of the **join** statement by:

return(*resource – invocation*)

The **return** statement in this context does not immediately return the results on completion of the resource access, but simply indicates the objects to be returned when the serializer operation terminates.

Figure 5 presents a simple serializer solution to a readers-writers problem discussed in [22], [23]. In addition to readers excluding writers and writers excluding other writers, the problem requires that if a read operation starts before a write operation, the reader will access the resource before the writer, and requires that this first-in-first-out ordering also be imposed on writers with respect to readers, and on writers with respect to other writers.

Simple serializers have fairly limited power. There are a number of extensions to simple serializers that would be of great utility, but most of these extensions require significant further research. Among these extensions are the addition of simple boolean variables and expressions for use in conditions, and the addition of conditional statements and looping constructs.

7.2. Serializer Semantics

Informally, the text of a serializer is a set of statements describing what happens when serializer operations are executed in a system with concurrent processes. More formally, each operation of a simple serializer can be represented by a sequence of *nodes*, each node representing a point in the operation where a change

```
FIFO = serializer is create, read, write

rep = record    [rc: crowd,                 % reader crowd
                 wc: crowd,                 % writer crowd
                 xq: queue,                 % common queue
                 rs: resource]              % resource

create = proc () returns (cvt)
          return(rep${rc: crowd$create(),
                      wc: crowd$create(),
                      xq: queue$create()
                      rs: resource$create()})
          end create

read = proc (ser: cvt, k: key) returns (value)
          % wait until there are no active writers
          enqueue ser.xq until crowd$empty(ser.wc)
          % become an active reader and perform the read
          join ser.rc do return(resource$read(ser.rs, k)) end
          end read

write = proc (ser: cvt, k: key, v: value)
          % wait until there are no active readers or writers
          enqueue ser.xq until crowd$empty(ser.rc) & crowd$empty(ser.wc)
          % become sole writer and perform the write
          join ser.wc do resource$write(ser.rs, k, v) end
          end write

end FIFO
```

Figure 5: FIFO Serializer

in the state of the serializer will occur. The state of a simple serializer consists of the state of the queues (excluding the external queue), the state of the crowds, and the state of serializer possession. There are six kinds of nodes:

1) **enter**(*name*): This node represents the initial entry to operation *name*. After executing this node, a process has possession of the serializer.

2) **enqueue**(*queue*, *condition*): This node represents the start of an **enqueue** statement. Executing this node places the process in the specified queue with the specified condition and releases possession.

3) **dequeue**(*queue*, *condition*): This node represents the end of an **enqueue** statement. Executing this node regains possession and removes the process from the specified queue.

4) **join**(*crowd*): This node represents the start of a **join** statement. Executing this node places the process in the specified crowd and releases possession.

5) **leave**(*crowd*): This node represents the end of a **join** statement. Executing this node regains possession and removes the process from the crowd.

6) **exit**: This node represents the termination of an operation. Executing this node releases possession.

An *event* represents the completion of execution of a node by a process. Events are atomic and take no time to occur, although there is a finite amount of time between events. An event consists of a *transaction* identifier and a node, where a transaction is the finite sequence of events that occur for a process in the execution of a serializer operation for a particular serializer object. The events in a transaction are in execution order.

A *history* is a (possibly infinite) sequence of events representing some interleaving of the (possibly incomplete) transactions involving a serializer object. For a given serializer object, there are infinitely many possible histories, depending on the requests made of that object and on the arbitrary choices possible in selecting **dequeue** events when several queues are ready. The semantics of a serializer is defined by stating which histories can be produced for any given serializer object. This is expressed in terms of a predicate which, given a representation of the serializer code and a history, returns true if and only if the history could be produced by the serializer. A history satisfying the predicate is called a *legal* history for the serializer. A legal history with no incomplete transactions is called a *complete* history.

7.3. Specification Language

The specification language for serializers is composed of clauses in which certain relations between events imply other relations between events. The meaning of specification clauses is given by stating rules for transforming the clauses into predicates on histories. The code for a serializer is said to meet its specifications if every complete history satisfies all of the specification predicates that result from the specification clauses for the code.

Only four classes of properties are addressed by the specification language:

Exclusion: one kind of access excludes another.

Priority: one transaction is served preferentially over another.

Concurrency: some accesses are required to be served concurrently.

Service: some (or all) accesses are required to run to completion.

Of course, not all interesting synchronization properties fall into these classes, although many do. For example, performance characteristics cannot be specified. Nor can all properties in these classes be expressed in the specification language.

The syntax of the specification language is given in Figure 6. An *event – symbol* is a transaction identifier, followed by the event kind, followed by optional information indicating other components of the event. A transaction identifier is given by the first letter(s) of the operation name, followed by optional digits if more than one transaction for that operation is referenced in the clause. A *node – symbol* identifies a particular node in an operation, and is given by the first letter(s) of the operation name, followed by a "*", followed by the event kind. Examples of event and node symbols are:

R-enter W1-join W2-leave
R*-enqueue W*-exit R*-dequeue

An *ordering* clause specifies a time ordering of events. A clause of the form GX(*e1*, *e2*, *e*) specifies that event *e* cannot occur between events *e1* and *e2*. A clause of the form GX(*e1*, *e2*, *n*) specifies that no events for node *n* can occur between events *e1* and *e2*. A clause of the form @*e* specifies that event *e* occurs in the history.

```
Clause   =   Clause "⊃" Clause
         |   Clause "&" Clause
         |   Clause "|" Clause
         |   "~" Clause
         |   Ordering
         |   "GX" "(" Event – symbol "," Event – symbol "," Event – symbol ")"
         |   "GX" "(" Event – symbol "," Event – symbol "," Node – symbol ")"
         |   "@" Event – symbol

Ordering =   Event – symbol "<" Event – symbol
         |   Event – symbol "<" Ordering
```

Figure 6: Specification Language Syntax

As an example, in readers-writers problems readers exclude writers, writers exclude readers, and writers exclude other writers. This can be specified as:

(R-join < W-join ⊃ R-leave < W-join) &
(W-join < R-join ⊃ W-leave < R-join) &
(W1-join < W2-join ⊃ W1-leave < W2-join)

A writer's priority specification of the form "if a reader and a writer enter the serializer while some other writer is being serviced, then the writer will be serviced before the reader" can be formalized as:

(W1-join < W-enter < W1-leave & W1-join < R-enter < W1-leave)
⊃ W-join < R-join

Concurrency for readers can be specified as:

GX(R1-enter, R2-enter, W*-enter) & R2-enter < R1-leave
⊃ R2-join < R1-leave

Finally, the service specification for readers and writers is:

(@R-enter ⊃ @R-exit) & (@W-enter ⊃ @W-exit)

7.4. Automatic Verifier

In proving that the code for a serializer meets its specifications, we need to state intermediate propositions about the serializer code and the specification clauses. Thus we require a language in which to state such propositions, and rules of inference for proving them. The proposition language used is an extension to the specification language described above. The extensions include more general event and node symbols, and a clause for dealing with serializer possession. Details of the extensions, and of the particular inference rules used, can be found in [20].

The input to the verifier consists of a description of the node sequence for each serializer operation and the specifications for the serializer. A typical specification clause is an implication consisting of a precondition and a consequent. Proving such a clause usually involves assuming the precondition is true and using the inference rules to derive the consequent.

Internally, the verifier uses an assertion stack containing all specification clauses that are proven true, as well as those that are assumed to be true. Paired with each clause in this stack are the inference rules that were used to assert the truth of the clause. This stack provides a record of which rules led to particular event orderings, as well as an efficient mechanism for removing assertions. Whenever a clause is inserted in the assertion stack, a check is made to determine if any additional

clauses can be immediately inferred. If an inference rule is applicable, the assertions implied by the rule are also added to the assertion stack. This, in turn, may lead to further inferences. The process completes when no further inference rules apply.

Before examining the specification clauses, the verifier performs a static analysis of each serializer operation. The node sequence for each operation is examined to determine when a transaction will have possession of the serializer, and when events are excluded because of conditions being false. For example, if a condition for a queue is *crowd$empty(c)*, where *c* is a crowd, then a **dequeue** event with that condition is prohibited from occurring between a **join** and a **leave** event for any transaction for that crowd. Each assertion derived from this analysis is added to the assertion stack.

After this static analysis, an attempt is made to prove each specification clause. For a specification clause of the form $P \supset Q$, the clause P is added to the assertion stack, and a check is made to see if Q is automatically inferred by the process described above.

If a consequent cannot be inferred directly, further methods must be used. It may be necessary to prove that a particular condition does or does not hold. Such a proof is usually done by contradiction, and may involve the introduction of anonymous transactions. For example, in proving that a crowd is empty, it is not sufficient to prove that none of the transactions in the transaction stack are in the crowd; it might still be possible for some unnamed transaction to be in the crowd. Therefore, an anonymous transaction is placed in the transaction stack and is assumed to be in the crowd, in the hope of inferring a contradiction.

Another method used by the verifier is proof by cases. When the relative order of **enter** events is not known, all possible orderings are tried. If a clause can be proven for every such ordering, the overall proof is established.

Finally, the verifier must handle service specifications such as

@R-enter $\supset$ @R-exit

Given that every unsynchronized resource invocation terminates, proof of service is accomplished by proving that each **dequeue** event a transaction can execute is guaranteed to occur. This proof is done largely by contradiction. First a **dequeue** event is assumed not to occur. This implies that the queue corresponding to the event is not empty, and that any crowds requiring **dequeue** events from that queue will eventually empty. This is generally enough information to prove that the condition for the **dequeue** event will eventually be satisfied.

To prove that the **dequeue** event eventually occurs, a fictitious "quiet point" event *QP* is introduced and proved to exist. This event has the property that it gains possession of the serializer only when no queues are ready, and occurs late enough in the history so that all of the necessary crowds have emptied. If the condition for the **dequeue** event holds at *QP*, and there can be no other process earlier in the given queue, then the **dequeue** event must precede *QP*, and the proof is complete. The existence of *QP* is established by proving that no other queues are ready at that point in the history.

8. SEMAPHORE PRIMITIVES AND STARVATION-FREE MUTUAL EXCLUSION

Semaphores were first introduced by Dijkstra as programming language primitives for synchronizing concurrently-executing processes. Most discussions of semaphore primitives in the literature give only informal descriptions of their behavior, rather than precise definitions. These informal definitions may be incorrect, incomplete, or subject to misinterpretation. As a result, the literature actually contains several different definitions of the semaphore primitives. The differences are important, since the particular choice of definition can affect whether a solution to the mutual exclusion problem allows the possibility of process *starvation*. Stark [24] has attempted to dispel some of the confusion by giving precise definitions of two varieties of semaphore primitives, which he calls *weak* and *blocked-set* primitives. He then shows that under certain natural conditions, although it is possible to implement starvation-free mutual exclusion with blocked-set semaphores, it is not possible to do so with weak semaphores. Thus weak semaphores are strictly less "powerful" than blocked-set semaphores.

Dijkstra [25] defines a semaphore as a special type of program variable, shared between processes, which may be manipulated only by two special operations, designated **P** and **V**. A semaphore variable may take on only nonnegative integer values. His definition of the effect of the semaphore operations is as follows: A process performing a **P** operation on a semaphore variable *s* tests the value of *s* to see if it is greater than zero. If so, then *s* is decremented and the process proceeds. The test and decrement are performed in one indivisible step. If the value of *s* is not greater than zero, the process is said to become *blocked* on the semaphore *s*, and must wait to be signalled by some process executing a **V**(*s*) operation. A process executing a **V**(*s*) operation checks to see if there are any processes blocked on *s*. If there are blocked processes, one of them is signalled and allowed to proceed. If there are no blocked processes, then *s* is simply incremented. The **V**(*s*) operation is assumed to be performed in a single indivisible step.

In Dijkstra's definition of semaphores, processes that are blocked within a **P** operation on a semaphore variable *s* are distinguished from processes that are "about" to execute a P(*s*), but have not yet become blocked. This distinction is important in that the execution of a V(*s*) will cause a blocked process to be selected in preference to a process that is not blocked. However, all blocked processes are treated equally as far as being selected is concerned: no effort is made to distinguish processes that have been blocked for a short length of time from those that have been blocked for a longer period. The group of blocked processes may therefore be modeled as a *set*, from which a **V** operation chooses at random a process to be signalled. Let us call semaphores with this type of blocking discipline *blocked-set* semaphores. It is also possible to define *blocked-queue* semaphores, which are like blocked-set semaphores except that the group of blocked processes is maintained as a FIFO *queue*, instead of as a set. Processes becoming blocked are placed at the end of the queue, and processes are selected for signalling from the head of the queue.

Another, much different, definition of semaphores is also found in the literature. This is the type of semaphore that may be implemented with indivisible "test-and-set" instructions as follows: A process attempting to perform a **P** operation on a semaphore variable *s* executes a busy-waiting loop in which the value of *s* is continually tested. As soon as *s* is discovered to have a value greater than zero, it is decremented, the decrement and immediately preceding test being performed as one indivisible step. A **V** operation simply increments *s* in an indivisible step. We will call this type of semaphore a *weak* semaphore.

Dijkstra also distinguishes between *binary* and *general* semaphores. In the discussion above, we informally defined weak, blocked-set, and blocked-queue *general* semaphores. *Binary* semaphores are similar to general semaphores except that the binary semaphore variable may take on only the values zero and one. The effect of a binary **P** operation is identical to that of a general **P** operation. However, to ensure that the value of the variable *s* never exceeds one, a binary V(*s*) operation will simply set *s* to one, rather than incrementing *s* as is done in a general V(*s*) operation. Note that if the value of a binary semaphore variable *s* is one, which implies that there are no processes blocked on *s*, then execution of a V(*s*) operation has no effect.

The weak and blocked-set semaphore primitives defined above have different starvation properties. To see why this might be true, let us see what happens when each definition is used in a simple attempt to solve the mutual exclusion problem. Consider a number of processes, each executing the following program:

```
        semaphore s initially 1;

loop:   <noncritical region>
        P(s);
        <critical region>
        V(s);
        goto loop;
```

Each process continually alternates between its *critical region* and its *noncritical region*. In order to ensure that mutual exclusion among all the processes is obtained, the critical region is bracketed by a P(*s*)-V(*s*) pair. Since the value of the semaphore variable *s* is initially one, and a process desiring to enter the critical region must first perform a P(*s*) operation, whenever some process is in its critical region, the value of *s* is zero. Hence other processes attempting to perform P operations and enter their own critical regions must wait. Mutual exclusion is therefore obtained regardless of whether weak, blocked-set, or blocked-queue semaphores are used.

Suppose that the semaphore operations are of the weak variety, and consider the execution of two processes, process 1 and process 2. Suppose that process 1 finds the value of *s* to be one, and proceeds into its critical region. Since the value of *s* is now zero, process 2 is unable to complete its P(*s*) operation, and therefore waits within the P operation for the value of *s* to become positive. Now suppose that process 1 completes execution in its critical region, and performs the V(*s*) operation, setting *s* to one. Since we have assumed the semaphore operations to be weak, process 2 does not complete its P(*s*) operation immediately, but must retest the semaphore variable *s*. It is possible, if process 1 executes quickly enough, for it to loop around and perform another P(*s*) operation, resetting *s* to zero, *before process 2 could get around to noticing that s ever had the value one*. This scenario may continue indefinitely, with the result that process 2 "starves" forever within its P(*s*) operation. Note that this argument relies on the fact that, in determining the behavior of a system of concurrent processes, we may make no assumptions about the relative speeds of the processes, and must consider all possible orders of executions of steps of the processes as legitimate.

Now, suppose instead that the semaphore operations are defined to be blocked-set operations. The scenario described in the preceding paragraph is no longer possible, since the execution of a V operation by process 1 immediately causes process 2 to complete its P(*s*) operation. Since *s* is never set to one, it is not possible for process 1 to complete another P(*s*) before process 2 finishes its critical region and performs a V(*s*). However, although starvation is no longer possible with two processes, with three or more processes it again becomes possible for a

process to wait forever within the P(*s*) while other processes successfully complete infinitely many P(*s*) operations. The reason for this is that the blocked-set V operation selects the blocked process to signal at random, and in particular, gives no preference to a process that may have been blocked for a long time. This situation may be remedied if blocked-queue semaphores are used.

The simple scenario just presented indicates that, although weak, blocked-set, and blocked-queue semaphores are all able to implement mutual exclusion of critical regions, the three types of semaphores are evidently not equivalent if the possibility of starvation is taken into consideration. To obtain more detailed information concerning the relative "power" of different kinds of semaphore primitives, Stark has developed a state-transition model of concurrently-executing sequential processes. Within this model, it is possible to give precise specifications for the behavior of various kinds of semaphore operations, as well as a precise definition of what it means to solve the "starvation-free mutual exclusion problem."

With these definitions, it becomes meaningful to investigate the relative "power" of weak and blocked-semaphores for implementing starvation-free mutual exclusion. Stark poses the question, "Are there solutions to the starvation-free mutual exclusion problem using only weak or only blocked-set semaphores?" The answer to this question is trivially "Yes," since there are solutions to the starvation-free mutual exclusion problem that do not make use of semaphores at all. To eliminate semaphore-free solutions from consideration, Stark defines some properties characteristic of "good" semaphore solutions to the starvation-free mutual exclusion problem, but which are not shared by the semaphore-free solutions. These properties are called *symmetry*, *no busy-waiting*, and *no memory*. He is then able to show that there are no weak semaphore solutions to the starvation-free mutual exclusion problem with all three of these properties, although such solutions exist using blocked-set semaphores. Thus blocked-set semaphores are more "powerful" than weak semaphores.

Stark presents two types of results: "positive" results, which assert the existence of certain solutions to the starvation-free mutual exclusion problem, and "negative" results, which assert the nonexistence of solutions satisfying various properties. Negative results are proved by assuming the existence of a solution satisfying the given properties and then obtaining a contradiction. The truth of the positive results is argued by actually displaying a solution with the desired properties. Stark gives a moderately formal proof of correctness for the most intricate of these solutions. The length of such proofs prohibits him from supplying more than informal correctness arguments for the other solutions.

The major contribution of this work is that it brings the murky issues of "fairness"

often mentioned in the synchronization literature into sharper focus. The attempt at precise definitions of the two types of semaphores helps to clear up confusion that has resulted from informal discussion. Many pages have been, and continue to be, spent in the literature in arguments over whether one program solves or does not solve a particular synchronization problem. Often such arguments are useless, since precise specifications are lacking, both for the synchronization problem itself, and for what it means to "solve" that problem. It is hoped that this work makes a small step toward the resolution of this difficulty.

References

1. Liskov, B. H., Snyder, A., Atkinson, R. R., and Schaffert, J. C. "Abstraction mechanisms in CLU," Communications ACM 20, 8 (August 1977), 564-576.

2. Liskov, B. H., Moss, J. E., Schaffert, J. C., Scheifler, R. W., and Snyder, A. "CLU reference manual," MIT/LCS/TR-225, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

3. Brinch Hansen, P. "Distributed processes: A concurrent programming concept," Communications ACM 21, 11 (November 1978), 934-941.

4. Hoare, C. A. R. "Communicating sequential processes," Communications ACM 21, 8 (August 1978), 666-677.

5. "Preliminary ADA reference manual," SIGPLAN Notices 14, 6 (June 1979).

6. Feldman, J. A. "High level programming for distributed computing," Communications ACM 22, 6 (June 1979), 353-368.

7. Randell, B. "System structure for software fault tolerance," IEEE Transactions on Software Engineering 1, 2 (June 1975), 220-232.

8. Shrivastava, S. K. and Banatre, J. P. "Reliable resource allocation between unreliable processes," IEEE Transactions on Software Engineering 4, 3 (May 1978), 230-240.

9. Roberts, L. G. and Wessler, B. D. "Computer network development to achieve resource sharing," Proceedings of the AFIPS 1979 Spring Joint Computer Conference, May 1970, 543-549.

10. Clark, D. D. et al. "An introduction to local area networks (draft)," Computer Systems Research Division, Request for Comments 163, MIT, Laboratory for Computer Science, Cambridge, Ma., April 1978.

11. Fuller, S. H. et al. "A collection of papers on CM*: A multi-microprocessor computer system," Carnegie-Mellon University, Department of Computer Science, Pittsburgh, Pa., February 1977.

12. Hewitt, C. "Viewing control structures as patterns of passing messages," Artificial Intelligence 8, 3 (June 1977), 323-364.

13. Liskov, B. H. "Primitives for distributed computing," Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, 33-42. Also Computation Structures Group Memo 175, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

14. Redell, D. D. "Naming and protection in extendible operating systems," MIT/LCS/TR-140, MIT, Laboratory for Computer Science, Cambridge, Ma., November 1974.

15. Herlihy, M. "Transmitting abstract values in messages," MIT/LCS/TR-234, MIT, Laboratory for Computer Science, Cambridge, Ma., May 1980.

16. Herlihy, M. and Liskov, B. "Communicating abstract values in messages," Computation Structures Group Memo 200, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1980.

17. Lampson, B. and Sturgis, H. "Crash recovery in a distributed data storage system," Xerox PARC, Palo Alto, CA, April 1979, unpublished.

18. Gray, J. "Notes on data base operating systems," in Operating Systems, An Advanced Course, Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

19. Kapur, D. "Towards a theory for abstract data types," MIT/LCS/TR-237, MIT, Laboratory for Computer Science, Cambridge, Ma., June 1980.

20. Atkinson, R. R. "Automatic verification of serializers," MIT/LCS/TR-229, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

21. Hewitt, C. and Atkinson, R. R. "Synchronization in actor systems," Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, Ca., January 1977, 267-280.

22. Courtois, P., Heymans, F., and Parnas D. "Concurrent control with readers and writers," Communications ACM 14, 10 (October 1971), 667-668.

23. Greif, I. "Semantics of communicating parallel processes," MIT/LCS/TR-154, MIT, Laboratory for Computer Science, Cambridge, Ma., September 1975.

24. Stark, E.W. "Semaphore primitives and starvation-free mutual exclusion," MIT/LCS/TM-158, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

25. Dijkstra, E. W. "Cooperating sequential processes," in Programming Languages, Genuys, F. (Ed.), Academic Press, 1968, 43-112.

Publications

1. Atkinson, R. R. "Automatic verification of serializers," MIT/LCS/TR-229, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

2. Berzins, V. "Abstract model specifications for data abstractions," MIT/LCS/TR-221, MIT, Laboratory for Computer Science, Cambridge, Ma., July 1979.

3. Bloom, T. "Evaluating synchronization mechanisms," Proceedings of the 7th Symposium on Operating Systems Principles, Pacific Grove, Ca., December 1979, 24-32. Also MIT, Laboratory for Computer Science, Computation Structures Group Memo 187, Cambridge, Ma., December 1979.

4. Herlihy, M. "Transmitting abstract values in messages," MIT/LCS/TR-234, MIT, Laboratory for Computer Science, Cambridge, Ma., May 1980.

5. Kapur, D. "Towards a theory for abstract data types," MIT/LCS/TR-237, MIT, Laboratory for Computer Science, Cambridge, Ma., June 1980.

6. Kapur, D. and Mandayam, S. "Expressiveness of the operation set of a data abstraction," Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, January 1980, 139-153.

7. Liskov, B. H. "Modular program construction using abstractions," Computation Structures Group Memo 184, MIT, Laboratory for Computer Science, Cambridge, Ma., September 1979. Also in Abstract

Software Specifications, 1979 Copenhagen Winter School Proceedings, Lecture Notes in Computer Science 86, Springer-Verlag, 1980, 354-389.

8. Liskov, B. H. "Primitives for distributed computing," Proceedings of the 7th Symposium on Operating Systems Principles, Pacific Grove, Ca., December 1979, 33-41. Also Computation Structures Group Memo 175-1, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

9. Liskov, B. H., Atkinson, R. R., Bloom, T., Moss, E., Schaffert, C., Scheifler, R., and Snyder, A. "CLU reference manual," MIT/LCS/TR-225, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

10. Liskov, B. H. and Snyder, A. "Exception handling in CLU," IEEE Transactions on Software Engineering SE-5, 6 (November 1979), 547-558.

11. Peterson, J. L. "Notes on a workshop on distributed computing," ACM Operating Systems Review 13, 3 (July 1979), 18-30.

12. Stark, E. "Notes on using TOPS-20," Computation Structures Group Memo 189, MIT, Laboratory for Computer Science, Cambridge, Ma., February 1980.

13. Stark, E. "Semaphore primitives and starvation-free mutual exclusion," MIT/LCS/TM-158, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

14. Stark, E. "Semaphore primitives and starvation-free mutual exclusion," Computation Structures Group Memo 194, MIT, Laboratory for Computer Science, Cambridge, Ma., June 1980.

Theses Completed

1. Allen, M. D. "A comparative analysis of programming languages," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., September 1979.

2. Atkinson, R. R. "Automatic verification of serializers," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., March 1980.

3. Berzins, V. "Abstract model specifications for data abstractions," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., July 1979.

4. Cooper, G. "An implementation of stable storage," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

5. Ehrenfried, D. "Design of a distributed calendar system," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

6. Herlihy, M. "Transmitting abstract values in messages," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., Aprii 1980.

7. Kapur, D. "Towards a theory for abstract data types," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., March 1980.

8. Stark, E. "Semaphore primitives and starvation-free mutual exclusion," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., March 1980.

Theses in Progress

1. Humphries, T. "Overloading in programming languages with data abstractions," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected February 1981.

2. Moss, E. "Distributed program environment," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, expected March 1981.

3. O'Connor, B. "A CLU code generator for the VAX 11/780," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, expected June 1981.

4. Schaffert, J. C. "The specification and proof of data abstractions in object-oriented languages," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, expected December 1980.

Talks

1. Bloom, T. "Evaluating synchronization mechanisms," Seventh Symposium on Operating Systems Principles, Pacific Grove, Ca., December 10, 1979.

2. Liskov, B. H. "Primitives for distributed computing,"

   Xerox Research Center, Palo Alto, Ca., August 6, 1979;
   Seventh Symposium on Operating Systems Principles,
   Pacific Grove, Ca., December 10, 1979;
   Carnegie-Mellon University, Pittsburgh, Pa., March 5, 1980.

3. Liskov, B. H. "Introduction to CLU," 2nd Advanced Course on Computing Systems Reliability, Toulouse, France, September 1979.

4. Liskov, B. H. "An example of modular program development," 2nd Advanced Course on Computing Systems Reliability, Toulouse, France, September 1979.

5. Liskov, B. H. "Embedding data abstraction in programming languages," 2nd Advanced Course on Computing Systems Reliability, Toulouse, France, September 1979.

6. Liskov, B. H. "Semantics of remote invocation," Quality Software Workshop, Carmel, Ca., December 9, 1979.

7. Liskov, B. H. Session Leader on Personal Computers and Their User Interface, 7th Symposium on Operating Systems Principles, Pacific Grove, Ca., December 11, 1979.

8. Liskov, B. H. "Abstraction mechanisms in CLU," College of William and Mary, Williamsburg, Va., March 28, 1980.

9. Srivas, M. K. "Expressiveness of the operation set of a data abstraction," Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Ne., January 30, 1980.

PROGRAMMING TECHNOLOGY

Academic Staff

A. Vezza, Group Leader
M. S. Blank
J. C. R. Licklider
L. Hawkinson

Research Staff

M. S. Broos
D. L. Dill
M. Dornbrook
S. W. Galley
G. E. Kaiser
P. D. Lebling
S. S. Pinter
R. Sangal
M. I. Travers

Graduate Students

M. J. Caruso
W. J. Noss

Undergraduate Students

S. H. Berez
A. Ghaznavi
T. K. Johnson
P. C. Lim
C. A. Renaud
S. Schad

Support Staff

J. A. Janoff
J. L. Schoof

PROGRAMMING TECHNOLOGY

1. INTRODUCTION

The primary activities of the Programming Technology group this year have been:

1) The study of a government office which we believe to be prototypical for offices of its kind that does short, medium and long term planning.

2) Analysis of the study to develop requirements for a planning system that would aid the office in its planning task.

3) Design of an initial planning system for such a prototypical office.

4) Implementation of parts of the planning system to test design concepts.

2. STUDY OF A PROTOTYPICAL OFFICE

The main activities of the "Office" are:

1) Defining missions and objectives that fall within its charter. This involves identifying problems or needs that the Office has a charter to solve or fill. The professional staff of the Office accomplish this through informal discussions with other professionals in the field and through formal study groups composed of suitable professionals. Thus this activity is largely an intellectual one and places a requirement on the planning system to create, aggregate, transmit, categorize, store and provide retrieval capability of textual material such as memoranda and messages.

2) Planning the acquisition of resources to accomplish the mission or solve the problem. There is both an intellectual aspect to this phase of the planning system resulting in a requirement similar to the one stated above and a well structured procedural aspect that could be by a computer program. In particular the budget planning cycle appears to be a highly procedural process that might be aided by computerization.

3) The acquisition of a suitable team to carry out the mission of solving the problem is similar to the second activity in that it is both an intellectual activity and one that is highly procedural.

A not too surprising constraint on the office is that even in those activities that are procedural in nature, the Office is not self contained. It must deal with data bases, such as the financial data base, that are not directly under its control. Its use of these data bases, especially wherever the use involves modification of content, is subject to constraints applied by other offices. Eventually, procedures for controlling access may be codified and reduced to computer programs but we expect that, for some time, an organization that owns a data base will require positive human control over extra-organizational access, especially updating. That is one of several sources of a requirement for an interpersonal message system. Communication among computers and data bases requires an interprocess message passing system.

Though there are procedural aspects to the Office's activities they are quite complex and far from the rigid. For this reason we are attempting to capture these activities in a knowledge base. Through this mechanism we hope to make it easier to have the users participate in the evolution of the planning system, provide English language descriptions of the planning system's sections, and provide a more natural user interface.

3. ADVANCED MESSAGE SYSTEM

The preliminary design for the message system was completed. The system is intended to facilitate communication and planning among a group of users performing management tasks. In the design, users work at workstations, which are personal computers and, at the same time, nodes in a communications network. Some of the tasks involve communication with remote resources. Some involve access to data bases. Some require coordination with other users. Large volumes of messages in many media must be accommodated.

3.1. System Attributes

The system is designed to take advantage of processing power of advanced personal computers. The message system will be highly distributed in the manner of the Laurel system [1]. It will be distributed because the *processing* power is distributed among advanced *nodes* connected in a high-capacity local network, not because the data is inherently distributed.

The system is designed to be robust and modular. The distributed architecture will contribute to robustness. One node will request services from their nodes by *name* rather than by location. Services will be redistributed among nodes as needed by renaming the nodes. Communication among distributed resources will rely on a *message-based* asynchronous architecture rather than a connection-based one.

The system is designed to handle a large volume of messages. It will store messages in a data base on a large-capacity node, as do DMS [2] and COMSYS [4], and it will provide indexes into the data base for faster access, as does DMS.

The system is designed to avoid the consistency problems of a distributed data base, by distinguishing between the *original* and *copies* of a message. The user will not *need* to retain data between sessions anywhere but in a central storage facility. The user will be free to choose any available node for a session. This does not prevent the user from storing data on a personal disk. However, there is no requirement that he do so.

The system is designed to utilize storage on personal computers effectively. Users will keep copies of frequently used parts of the data base on their nodes in a cache-like strategy.

The system is designed to allow users to perform most operations locally to enhance privacy. Some users may wish to operate in a manner which minimizes use of the central data base and message processors. It will be possible to perform most operations entirely in the personal computers if the users so desire. We will investigate which operations are portable in this respect and which are not, and what costs in speed and storage efficiency are paid.

The system is designed to provide structured communication among separated programs. The nodes will transmit "dispatches" (MDL-like structures) in the highest protocol layer. Shortcuts like shared storage can be allowed when feasible, as long as the same protocol is followed, to allow redistribution of services. The familiar operation of message transmission will be one variety of dispatch.

The system design concerns itself as little as possible with lower layer communication issues. The design assumes that lower layers of protocol can guarantee error-free transmission and encryption if necessary.

The system is designed to provide typical communication services. The central node will provide distribution lists for multiple-point delivery. Users will be able to annotate messages and specify who (if anyone) can see the annotations. "Blind" copies will be supported. Messages can be printed in hard copy, transmitted through long-haul networks like ARPANET, and so on. Messages can be registered like land deeds and certified copies provided. Registration and certification require some kind of underlying encryption mechanism; we are not doing any original work in this area, we will use the best mechanisms available at the implementation time. Similarly, privacy in the system will rely on existing and available access-control mechanisms.

The system is designed to facilitate user interaction. The interactive language will

carry over into the area of use and control of programs the concepts of "high level abstraction" and "structural modularity" that have been developed significantly in programming languages. The (nontechnical) user will think and act at a high level, communicating *intentions* and *purposes* to his node. The system, supported by its knowledge bases, will route messages, translate the data base queries, and so on. The program in the node will act as the user's agent and define the details of the actions and the communications in such a way as to carry out the user's intent. The user interface is intended to be natural and easy to use. It will optionally employ something approaching natural language as well as graphics, voice and so on. A message can include text, images and sounds.

The system exploits the user's mental models. The user's mental model of the system's structure should be as similar as possible to that of his office in his organization. The apparent storage structure will mimic a typical office, as in DMS and READER [4]. The system will use a formalized taxonomy of office and organizational terms as a *lingua franca* of interaction between servers.

The system permits the user to act in different roles. It is common for a person in an organization to "wear different hats" at different times. The system will allow roles to have names and will allow restrictions on which persons can act in each role, as in DMS.

The system is designed so that its structure does not obtrude on the user's perceptions. The user will be reminded as little as possible that the system is distributed, that messages migrate to and from the user's node, and so on.

The system is designed to relieve the user of remembering office procedures. The knowledge base will store and utilize both formal and informal procedures.

3.2. Architecture

Communication between nodes is accomplished through several layers of protocols. These build up from the layer of the underlying communication network to the topmost structured protocol. We will build only the topmost layer, which will provide transmission of MDL-style structures called *dispatches*, with guaranteed delivery of each dispatch to a suitable node (perhaps one of many). We assume that lower protocol layers can guarantee error-free transmission to a particular node.

Most nodes are advanced "personal computers" such as Alto or Nu or Lisp Machine. These are referred to as *workstations*. A large-capacity node, called the *central node*, with the MIT-XX Tops-20 system as its initial realization, will provide many services for the system. A medium-capacity node, called IFS because the Ethernet IFS will be the prototype, will store dispatches destined for the central node when the latter is not available.

Servers are programs that are resident on various nodes and provide services.

Services include:

- Message distribution, "filtering" to produce different "views" of a message for different users, and message storage.

- Message registration (like registration of real estate deeds) and certification (providing unaltered "copies" of registered messages) also.

- Communication with various other data base systems, for example financial, is planned.

- Knowledge bases.

- Printing.

- Long-haul transmission (for example ARPANET).

The parts of the system will communicate in the common language of dispatches. Services will interface the language of dispatches to the outside world. Such interfacing will be invisible to the user and to the programs that work within the message system. For example, to interface to a data base that does not know about the system, there will be a service that handles all transactions with that data base. From the viewpoint of the message system, the service *is* the data base.

A *message* is either a record in a relational data base on the central node, or a copy of an apparent copy of another node. The central message data base is called the central *message store* (MS), and it is called "relational" because it is designed to use a "flat" storage scheme with one MDL object per message in the ASYLUM "file system".

Messages and references to messages are transmitted among users. If the large-capacity nodes are not available, the system must not grind to a halt. However, users must have the ability to ascertain that a given copy of a message is a "true copy". Because there is potentially no control over the software on the personal nodes, the only reliable source for a "true copy" of a message will be the central message store.

There will be at least two kinds of messages: *formal registered* (that is "registered" and "on the record") messages, and *informal* messages. Messages will be stored on other nodes to provide faster access, but these messages (whether copies of registered messages or nonregistered messages) will be potentially alterable by the node's owner and thus classified as "informal" messages.

Access to an unalterable "copy" of a message will be provided by a *citation* (or "pointer") to the actual message. Access controls can be stored with the message or citation or both. "Forwarding" a message means either sending a citation, which the receiver can use to get a view of the message from the central node, or sending a copy of the message itself, which the receiver should treat as "hearsay" or secondhand information.

Registered messages, by policy, cannot be altered. MS messages can only be appended to (and then only in certain fields), with each addition marked with the user and time of appending.

A message will be identified by its registration number (if any) or its MS accession number (if any) or a combination of sender, node, and time.

Messages will be transmitted between nodes either directly or by background processes.

3.3. Dispatches

The AMS is based on the concept of *dispatches* between *processes* A dispatch is a carrier of information; most of them are either requests for action or responses. Dispatches include sending messages, reading data from a data base, setting up an alert, updating a data base, and many others. Dispatches are performed by *services* which re part of message system *servers*. Each service is associated with one or more servers (for example, the service of message registration is provided by the *message demon* server on the central node). Dispatches are the method by which services communicate among themselves.

It is very important to distinguish a *dispatch* from a *message*. Dispatches are the structures by which message system operations are performed. Such operations *include* (but are not limited to) sending messages. Sending a message is one type of dispatch, and since this is a message system, many dispatch types are concerned with messages and message sending, but the concept of a dispatch encapsulates that of a message.

4. MACHINE INDEPENDENT MDL

The planning system project has the need for the development of a machine-independent implementation of the language MDL, which has been the Programming Technology group's primary development language for the past eight years [3] [5]. The reason for this is twofold:

1) Members of the laboratory have a great deal of experience with MDL and have used it successfully in the past in the development of large systems, including message and database systems. It has been suggested that parts of the newly designed system may be extracted from the MDL code running on DM machine's communications system [4].

2) Since the current project involves a distributed approach to a knowledge based message system, there is a great advantage to writing the system in such a way that any "module" (e.g., the knowledge base) can be transferred between any of a number of machines (e.g., the 2060 and the Nu terminal) with at most a minor change in code.

The aim, therefore, is to develop a machine-independent MDL which is generally compatible with the current MDL, and which should be able to perform well on a large number of machines. The approach which has been chosen is the development of an MDL virtual machine, running a language called MIM (Machine Independent MDL). This approach is analogous to that taken in the implementation of PASCAL, with the aid of "P-codes". Only a MIM interpreter must be written for each target machine. MIM code will be compiled from MDL code by a process to be called MIMC (the MIM Compiler), which will be written in MDL. It is also proposed that eventually, order-code compilers be written for each target machine to enhance efficiency, but this is not required on the various target machines. In the more distant future, consideration may be given to microcoding MIM instructions to realize the "M" machine in hardware.

There are four subprojects involved in the creation of a machine-independent MDL:

1) Design of the "M" Virtual Machine

2) Implementation of an "M" Virtual Machine for the 2060 initially and then for the Nu terminal

3) Writing the MIM Compiler (MIMC)

4) Writing of the MDL interpreter in MDL

At the present time, the design of the virtual machine is nearly complete and work is progressing on the 2060 virtual machine as well as the MDL interpreter written in MDL. It is projected that a prototype virtual machine running an MDL interpreter will be operational by the fall of 1980. A design note containing the specifications for the virtual machine is written, as SYS.18.01.

References

1. Levin, R. and Schroeder, M. "Transport of electronic messages through a network," Teleinformatics 79, North-Holland 1979, 29-33.

2. Broos, M., Galley, S. W. and Vezza, A. "The data-based message service," MIT, Laboratory for Computer Science, Technical Report (not yet published).

3. Galley, S. W. and Pfister, G. "The MDL Programming Language," MIT, Laboratory for Computer Science, Cambridge, Ma., 1979.

4. Lebling, P. D. "COMSYS/READER: An electronic communication system," MIT, Laboratory for Computer Science, Technical Report (not yet published).

5. Lebling, P. D. "The MDL Programming Environment," MIT, Laboratory for Computer Science, Cambridge, Ma., 1980.

Publications

1. Blank, Marc S. "Machine-independent MDL: A preliminary design note," MIT, Laboratory for Computer Science, Programming Technology Division Document, SYS.18.01, June 1980.

2. Kaiser, Gail E. "Automatic extension of an augmented transition network grammar for Morse code conversations," MIT/LCS/TR-233, MIT, Laboratory for Computer Science, Cambridge, Ma., March 1980.

3. Lebling, P. D. "The MDL Programming Environment," MIT, Laboratory for Computer Science, Cambridge, Ma., 1980.

Theses Completed

1. Hwang, A. H. "An input encoding scheme for Chinese characters," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1979.

2. Johnson, T. K. "Definition of forms for computerized form-based communication systems," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., September 1979.

Theses in Progress

1. Noss, Wayne J. "Database alerting mechanisms," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1980.

Talks

1. Licklider, J.C.R. "Technical opportunities and policy issues," Interagency Committee on Automatic Data Processing, January 25, 1980.

2. Licklider, J.C.R. "Information technology: Challenges of the 1980's," Williamsburg, Va., January 28, 1980.

3. Licklider, J.C.R. "Libraries and information technology: 1980-2000," Council of University Presidents of the Association of Universities and Colleges of Canada in Ottawa, Ontario, February 21, 1980.

4. Licklider, J.C.R. "Social and economic impacts of information technology on education," Congressional Hearings on Information Technology in Education, Washington, D.C., April 2, 1980.

5. Vezza, A. "Design issues for computer message systems," Telecommunications Training Program for the Business Communications Review, New York, December 6, 1979.

REAL TIME SYSTEMS

Academic Staff

S. A. Ward, Group Leader
M. L. Dertouzos
R. H. Halstead

Research Staff

J. Gula
R. McLellan

Graduate Students

C. Baker
C. Cesar
D. Goddeau
M. Johnson
A. Mok
W. Paseman
M. Patrick
J. Powell
A. Reuveni
J. Sieber
T. Teixeira
C. Terman
J. Troisi

Undergraduate Students

J. Arnold
J. Bakopoulos
E. Becker
W. Cattey
C. Eliot

Support Staff

L. Kenen
E. Tervo
L. Zolman

REAL TIME SYSTEMS

1. INTRODUCTION

Major research activities of the Real Time Systems group during the past year have been (i) continued development of the Nu personal computer; (ii) design and implementation of the TRIX operating system; (iii) continued investigation of the MuNet and similar scalable multiprocessor architectures; and (iv) new research in the area of VLSI design tools. The first three of these projects have been described in previous progress reports.

2. NU: THE LCS PERSONAL COMPUTER

During the past year an extensive redesign of the Nu hardware architecture was completed and released to Zenith for manufacture (J. Arnold, D. Goddeau, R. McClellan, C. Terman, S. Ward). An initial ten prototype Nu's have been delivered to LCS recently by Zenith, and are currently being readied for use within the Laboratory. These prototypes feature 1024-line bitmap displays, Motorola 68000 processors, 256 Kbyte primary memories, and local disk storage of 20 Mbytes (10 Mbytes fixed, 10 Mbytes removable). Interfaces to the LCS Network are under development, and are expected to be available for the prototype Nu's toward the end of the summer.

Our current relationship with Zenith has its roots in the manufacturing agreement between LCS and the Heath Company, mentioned in last year's report. Heath's acquisition by Zenith confused the latter agreement, necessitating the negotiation of a new contract (whose terms are incidentally more favorable to LCS than those of the Heath contract, due primarily to the bargaining wizardry of M. Dertouzos).

3. TRIX: A NETWORK-ORIENTED OPERATING SYSTEM

A TRIX kernel has been implemented for the 68000-based Nu machines (J. Gula). While this initial system (designated TRIX 0.0) is still shaky in a number of respects, it provides TRIX message-passing and process management sufficiently well to support a UNIX-like programming environment; we plan to distribute it (together with a UNIX-compatible file system developed by J. Sieber) to sympathetic Nu users within LCS during the summer. A C compiler (adapted by Terman from Bell's Portable C Compiler) will also be available as part of the initial support package, along with a variety of software bootstrapped from UNIX.

4. MUNET: SCALABLE MULTIPROCESSOR ARCHITECTURES

The anatomy of scalable multiprocessor architectures has been the subject of continuing research by R. Halstead in connection with the MuNet project. Such an architecture has the potential advantages of *flexibility*, in that one basic architecture can be made to satisfy a wide variety of demands by simple addition or deletion of modules, *reliability*, if failed modules can be isolated and their functions taken over by like modules elsewhere in the system, and *computational power* exceeding that of any currently existing machine, if the architecture proves to be scalable up to truly large sizes. In order for an architecture such as this to be useful, however, the problem of developing software for it must be tractable. To simplify this task, the MuNet architecture includes protocols that support a virtual machine in which all communication functions are completely transparent to the user, leaving him only the task of specifying algorithms sufficiently parallel to keep large numbers of modules all occupied concurrently.

The principal hardware facility that has been used for this research is the group's collection of LSI-11 microcomputers, which may be configured into a variety of network topologies. Work on the basic protocols necessary to implement a MuNet on this system was completed just about a year ago; the period since then has been one for study of the results obtained and construction of aids to the development of more interesting test applications for the MuNet, along with refinement and publication of some of the MuNet mechanisms. Notably, the virtual machine seen by programs executing on a MuNet has been characterized more formally. This in turn has led to the consideration of some alternative virtual machines with differing mechanisms for atomicity and synchronization. To a greater extent than has been true of earlier proposals, these machines are expressed at a level close to that of a microcoded implementation.

The coming year will see experimentation with these alternative virtual machines in the context of some more realistic applications. These, combined with refinements of the object management and load distribution schemes, should bring the MuNet design closer to the ultimate goal of a concrete implementation using VLSI techniques.

5. VLSI DESIGN AIDS

In the past year work was begun on a family of tools to be used in designing and testing Very Large Scale Integrated circuits. Our initial efforts have been directed towards building a set of tools that allow the designer to interactively verify the correctness of a completed design. All information used in the verification process is derived from the mask information as it will be sent to the manufacturer, thus if the

design successfully runs our gauntlet of tests, there is a high probability that the chip will work when built.

There are two stages in the testing process: *static tests* which look for syntactic errors in the masks, and *dynamic tests* which simulate the electrical circuit providing logic and timing information. The static tests are a series of programs that:

1) Check for adherence to design rules specifying the minimum line widths, spacing between features, etc.

2) Extract electrical network (nodes and transistors) from mask information. Resistance and capacitance of nodes and length/width ratios of transistors are also estimated.

3) Examine the network to ensure each node is well-formed: e.g., each node can be set to both "0" and "1," no double threshold drops, no shorts, etc.

4) Check pullup and pulldown ratios.

A main design goal of all the tools is an ability to process large designs and to provide fast turnaround (measured in at most hours) for the designer. The design rule checker and node extractor are based in part on a raster scan of the mask and so are potential candidates for implementing directly in hardware.

The basic logic simulation algorithm models the circuit under test as a network of transistor switches and pullups. Changes in the circuit inputs are propagated through the network, using an event-based scheduler to direct the computation. In order to efficiently simulate large and/or highly parallel circuits, a number of presimulation network transformations can be applied to condense subnetworks that implement logic gates, PLA's, registers, etc. into equivalent, more computationally tractable structures such as tables or compiled functions. A final modification to the original algorithm extends the basic unit-delay simulation to allow the switching delays to be calculated from the electrical characteristics of the appropriate nodes and transistors. The algorithms were incorporated in a simple, interactive simulation environment, and subsequently used to debug a number of designs at MIT, BBN, and Stanford. The current tools have sufficient power to handle large (> 10,000 transistors) designs with satisfactory turnaround times; future development work should allow much larger designs to be accommodated. Since the input networks are automatically derived from the mask files, successful simulations have proved to be harbingers of fully operational chips.

In the next year, we hope to integrate these tools into an interactive VLSI design

system to be implemented on Nu, the LCS personal computer. Ultimately, this system will incorporate a graphics editor for cell design using the high resolution raster scan graphics of the Nu; an embedded layout language and router used for cell interconnection; and a static checker/simulator (optionally supported with special hardware for increased performance).

Publications

1. Cesar, C. "Real-time emulation of hardware," Real Time Systems Group Working Paper 1979-5, MIT, Laboratory for Computer Science, Cambridge, Ma., August 1979.

2. Dertouzos, M. L. and Moses, J., Editors, The Computer Age: A Twenty Year View, MIT Press, Cambridge, Ma., 1979.

3. Halstead, R. "The VIM Virtual Machine for multiprocessor systems," Proceedings of Spring COMPCON (IEEE Computer Society), San Francisco, February 1980.

4. Halstead, R. and Ward, S. "The MuNet: A scalable decentralized architecture for parallel computation," Proceedings of the Seventh International Symposium on Computer Architecture, La Baule, France, May 1980.

5. Mok, A. and Ward, S. "A distributed broadcast channel access," Computer Networks 3, 5 (November 1979), 324-335.

6. Ward, S. and Halstead, R. "A syntactic theory of message passing," JACM 27, 2 (April 1980), 365-383.

7. Ward, S. "An approach to real-time computation," to be published in IEEE Transactions on Software Engineering.

8. Ward, S. "TRIX: A network-oriented operating system," invited paper, Proceedings of Spring COMPCON (IEEE Computer Society), San Francisco, February 1980.

9. Ward, S. "Architecture of a scalable computer network node," Proceedings of Spring COMPCON (IEEE Computer Society), San Francisco, February 1980.

Talks

1. Baker, C. "Network topology via node extraction," MIT VLSI Research Review, Cambridge, Ma., May 1980.

2. Dertouzos, M. "Computers and people: Some Delphi-oracle predictions," Invited lecture for the 1980 Zinon Papanastassiou Memorial Lectureship, Hellenic College, Brookline, Ma., April 1980.

3. Dertouzos, M. "Computers and people in the next twenty years," Invited lecture for the MIT Industrial Liaison Program, Tojo Kaikan, Tokyo, Japan, March 11, 1980.

4. Dertouzos, M. "Some expected computer developments and consequences: The next two decades," Testimony before the U.S. Senate Judiciary Committee on the Federal Computer Systems Protection Act of 1979, Washington, D. C., February 28, 1980.

5. Dertouzos, M. "Societal prospects and problems of information processing," Invited lecture, International Symposium on Computer, Man and Society, Haifa, Israel, October 23, 1979.

6. Halstead, R. "The MuNet: A scalable decentralized architecture for multiprocessor systems,"

   Cii Honeywell Bull, Paris, France, May 1980;
   Brown Boveri Research Laboratories, Baden, Switzerland, May 1980.

7. Mok, A. "Fundamental design problems of multiprocessor systems for hard real-time environments,"

   Duke University, Durham, N. C., March 24, 1980;
   Naval Research Laboratory, Washington, D. C., May 13, 1980.

8. Terman, C. "Compiling programs for a real time environment,"

   Bell Labs, Murray Hill, N. J., November 1979;
   General Electric, Schenectady, N. Y., January 1980;
   Intel, Aloha, Oregon, February 1980.

9. Terman, C. "Verification of VLSI designs,"

Digital Equipment Corp., Westboro, Ma., March 1980;
Intel, Santa Clara, Ca., June 1980.

10. Terman, C. "An approach to personal computing," COMPCON, San Francisco, Ca., February 1980.

11. Ward, S. "An approach to personal computing," CMU Computer Science Seminar, Carnegie-Mellon University, Pittsburgh, Pa., September 1979.

12. Ward, S. "Nu: A personal computer system," Bell Laboratories (videotaped for limited distribution), Holmdel, N.J., October 1979.

13. Ward, S. "The architecture of a scalable computer system," Hewlett-Packard, Palo Alto, Ca., October 1979.

14. Ward, S. "TRIX: An operating system for a distributed computer system," XTREE Seminar, Univ. Calif. at Berkeley, Ca., October 1979.

15. Ward, S. "NU: An approach to personal computing," New Horizons in Physics Lecture Series, SUNY, New Paltz, N.Y., November 1979.

Theses Completed

1. Baker, C. "Artwork analysis tools for VLSI circuits," S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

2. Bakopoulos, J. "The journal system," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

3. Buchan, P. "A full screen user interface for DARTS," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

4. Dennison, L. "An application of multiple processes to file management systems," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

5. Fuccio, M. "A modular language system for a multiprocessor," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

6. Hsia, C. "The design and implementation of a floating point processor," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

7. Patrick, M. "Mixing interpreted and executed program environments: Issues and problems," S.B./S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

8. Weihl, W. "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," S.B./S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

Theses in Progress

1. Eliot, C. "An automatic telephone directory and dialer," S.B. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

2. George, P. "Performance analysis of real-time systems," S.B./S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1980.

3. Mok, A. "Fundamental design problems of multiprocessor systems for hard real-time environments," Ph.D. dissertation, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

4. Paseman, W. "Some new methods of music synthesis," S.M. thesis, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected June 1980.

5. Terman, C. "Compiling programs for a real-time environment," Ph.D. dissertation, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected August 1980.

6. Teixeira, T. "Radical optimizations in real-time programs," Ph.D. dissertation, MIT, Dept. of Electrical Engineering and Computer Science, Cambridge, Ma., expected August 1980.

TECHNICAL SERVICES

Research Staff

K. T. Pogran, Group Leader

Undergraduate Students

C. Ludwig

S. Routhier

Support Staff

R. G. Bisbee
O. Feingold
J. Ricchio
T. Sealy

TECHNICAL SERVICES

1. INTRODUCTION

The Technical Services group functions in a support role within the Laboratory. In this capacity it provides:

1) Support for designing and prototyping the LCS Local Ring Network hardware (LNI). The development of the LNI was carried out by the Computer Systems Research group and the results of that effort are reported in that section of this report.

2) Maintenance support for the Laboratory's approximately 170 terminals and personal computers.

3) A TIP liaison function.

4) Maintenance of an electronics laboratory facility for use by all LCS groups.

5) Management and maintenance of the LCS network.

6) Management of the Laboratory computer operations including file backup and retrieval.

2. NEW ACTIVITIES

The Technical Services group installed 19 Xerox Altos, an Ethernet, a network file server and high quality Dover laser printing device. The group has taken over the hardware maintenance support of the Altos, their associated disk drives, and the Ethernet. The Ethernet was also connected to the Artificial Intelligence Laboratory PDP-11 gateway to the Chaos network and the LCS ring network and the AI Chaos network. Thus a physical data path between the Xerox Dover and the PDP-10's, KL-10, and 2060 computers was established.

Four LNI's were installed (two in March 1979 and two in September 1979) at the University of California at Los Angeles (UCLA) establishing a four node two megabit ring network of PDP-11 computers for the UCLA security project.

DISTRIBUTED SYSTEMS THEORY

Academic Staff

H. Abelson

Graduate Students

P. Andreae

DISTRIBUTED SYSTEMS THEORY

The objective of this work is to develop techniques for understanding the inherent constraints on computations performed by distributed systems. One important class of such constraints arises from the need to share information between different parts of a system during the course of computation. In previous work, Professor Abelson defined the notion of total information transfer for computations performed in a distributed network, and developed general techniques for determining lower bounds on the total information transfer required for computing any of a wide class of numerical functions. These methods were applied in analyzing the information transfer requirements for solving large systems of linear equations in various network configurations. This work was reported in Abelson's paper on information transfer which appeared in Journal of the ACM this spring.

During the past year Abelson built upon these methods to establish the existence of and to study functions which are truly "indecomposable," i.e., functions whose computations require large amounts of information transfer, regardless of how the initial data is distributed among the processors of a system. He also studied analogues of the general lower bound methods which are applicable, not only to numerical functions, but to Boolean functions as well. The results for Boolean functions form the basis of a graduate thesis by Peter Andreae, under Abelson's supervision, to be completed this summer.

The most important advance during the past year was the realization that the information transfer techniques originally developed for studying distributed networks, are also directly applicable to the derivation of inherent area-time trade-offs for VLSI circuits. Using these techniques, Abelson and Andreae were able to show that any n-bit multiplier chip must satisfy the constraint $AT^2 > n^2/64$, where A is the area of the chip and T is the time of operation. This was reported in a paper by Abelson and Andreae in Communications ACM.

During the next year Abelson proposes to continue investigating the role of information transfer, both for networks and for chips. He expects that the methods used to establish area-time trade-offs for chips can also be used to derive specific techniques, which deal with worst case performance to take account of average case performance. In addition, he intends to explore the implications of these lower bound methods for the design of algorithms and architectures which minimize information transfer.

References

1. Abelson, H. and Andreae, P. "Information transfer for VLSI multiplication," Communications ACM 23, 1 (January 1980), 20-23.

2. Abelson, H. "Lower bounds on information transfer in distributed computations," Journal of the ACM 27, 2 (April 1980), 384-392.

TM-65 Fischer, Michael J.
The Complexity Negation-Limited Networks - A Brief Survey, June 1975.

* TM-66 Leung, Clement
Formal Properties of Well-Formed Data Flow Schemas, S.B., S.M. & E.E. Thesis, EE Dept., June 1975.

* TM-67 Cardoza, Edward E.
Computational Complexity of the Word Problem for Commutative Semigroups, S.M. Thesis, EE & CS Dept., October 1975.

* TM-68 Weng, Kung-Song
Stream-Oriented Computation in Recursive Data Flow Schemas, S.M. Thesis, EE & CS Dept., October 1975.

* TM-69 Bayer, Paul J.
Improved Bounds on the Costs of Optimal and Balanced Binary Search Trees, S.M. Thesis, EE & CS Dept., November 1975.

* TM-70 Ruth, Gregory R.
Automatic Design of Data Processing Systems, February 1976.
AD A023-451

* TM-71 Rivest, Ronald
On the Worst-Case of Behavior of String-Searching Algorithms, April 1976.

* TM-72 Ruth, Gregory R.
Protosystem I: An Automatic Programming System Prototype, July 1976.
AD A026-912

* TM-73 Rivest, Ronald
Optimal Arrangement of Keys in a Hash Table, July 1976.

TM-74 Malvania, Nikhil
The Design of a Modular Laboratory for Control Robotics, S.M. Thesis, EE & CS Dept., September 1976.
AD A030-418

TM-75 Yao, Andrew C., and Ronald I. Rivest
K + 1 Heads are Better than K, September 1976.
AD A030-008

* TM-76 Bloniarz, Peter A., Michael J. Fischer and Albert R. Meyer
A Note on the Average Time to Compute Transitive Closures, September 1976.

* TM-77 Mok, Aloysius K.
Task Scheduling in the Control Robotics Environment, S.M. Thesis, EE & CS Dept., September 1976.
AD A030-402

* TM-78 Benjamin, Arthur J.
Improving Information Storage Reliability Using a Data Network, S.M. Thesis, EE & CS Dept., October 1976.
AD A033-394

* TM-79 Brown, Gretchen P.
A System to Process Dialogue: A Progress Report, October 1976.
AD A033-276

TM-80 Even, Shimon
The Max Flow Algorithm of Dinic and Karzanov: An Exposition, December 1976.

TM-81 Gifford, David K.
Hardware Estimation of a Process' Primary Memory Requirements, S.B. Thesis, EE & CS Dept., January 1977.

TM-82 Rivest, Ronald L., Adi Shamir and Len Adleman
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, April 1977.
AD A039-036

TM-83 Baratz, Alan E.
Construction and Analysis of Network Flow Problem which Forces Karzanov Algorithm to $O(n^3)$ Running Time, April 1977.

* TM-84 Rivest, Ronald L., and Vaughan R. Pratt
The Mutual Exclusion Problem for Unreliable Processes, April 1977.

* TM-85 Shamir, Adi
Finding Minimum Cutsets in Reducible Graphs, June 1977.
AD A040-698

TM-86 Szolovits, Peter, Lowell B. Hawkinson and William A. Martin
An Overview of OWL, A Language for Knowledge Representation, June 1977.
AD A041-372

* TM-87 Clark, David., editor
Ancillary Reports: Kernel Design Project, June 1977.

TM-88 Lloyd, Errol L.
On Triangulations of a Set of Points in the Plane, S.M. Thesis, EE & CS Dept., July 1977.

* TM-89 Rodriguez, Humberto Jr.
Measuring User Characteristics on the Multics System, S.B. Thesis, EE & CS Dept., August 1977.

* TM-90 d'Oliveira, Cecilia R.
An Analysis of Computer Decentralization, S.B. Thesis, EE & CS Dept., October 1977.
AD A045-526

* TM-91 Shamir, Adi
Factoring Numbers in O (log n) Arithmetic Steps, November 1977.
AD A047-709

* TM-92 Misunas, David P.
Report on the Workshop on Data Flow Computer and Program Organization, November 1977.

* TM-93 Amikura, Katsuhiko
A Logic Design for the Cell Block of a Data-Flow Processor, S.M. Thesis, EE & CS Dept., December 1977.

* TM-94 Berez, Joel M.
A Dynamic Debugging System for MDL, S.B. Thesis, EE & CS Dept., January 1978.
AD A050-191

* TM-95 Harel, David
Characterizing Second Order Logic with First Order Quantifiers, February 1978.

* TM-96 Harel, David, Amir Pnueli and Jonathan Stavi
A Complete Axiomatic System for Proving Deductions about Recursive Programs, February 1978.

TM-97 Harel, David, Albert R. Meyer and Vaughan R. Pratt
Computability and Completeness in Logics of Programs, February 1978.

TM-98 Harel, David and Vaughan R. Pratt
Nondeterminism in Logics of Programs, February 1978.

TM-99 LaPaugh, Andrea S.
The Subgraph Homeomorphism Problem, S.M. Thesis, EE & CS Dept., February 1978.

TM-100 Misunas, David P.
A Computer Architecture for Data-Flow Computation, S.M. Thesis, EE & CS Dept., March 1978.
AD A052-538

* TM-101 Martin, William A.
Descriptions and the Specialization of Concepts, March 1978.
AD A052-773

* TM-102 Abelson, Harold
Lower Bounds on Information Transfer in Distributed Computations, April 1978.

* TM-103 Harel, David
Arithmetical Completeness in Logics of Programs, April 1978.

* TM-104 Jaffe, Jeffrey
The Use of Queues in the Parallel Data Flow Evaluation of "If-Then-While" Programs, May 1978.

* TM-105 Masek, William J., and Michael S. Paterson
A Faster Algorithm Computing String Edit Distances, May 1978.

* TM-106 Parikh, Rohit
A Completeness Result for a Propositional Dynamic Logic, July 1978.

* TM-107 Shamir, Adi
A Fast Signature Scheme, July 1978.
AD A057-152

TM-108 Baratz, Alan E.
An Analysis of the Solovay and Strassen Test for Primality, July 1978.

* TM-109 Parikh, Rohit
Effectiveness, July 1978.

TM-110 Jaffe, Jeffrey M.
An Analysis of Preemptive Multiprocessor Job Scheduling, September 1978.

TM-111 Jaffe, Jeffrey M.
Bounds on the Scheduling of Typed Task Systems, September 1978.

* TM-112 Parikh, Rohit
A Decidability Result for a Second Order Process Logic, September 1978.

TM-113 Pratt, Vaughan R.
A Near-optimal Method for Reasoning about Action, September 1978.

* TM-114 Dennis, Jack B., Samuel H. Fuller, William B. Ackerman, Richard J. Swan and Kung-Song Weng
Research Directions in Computer Architecture, September 1978.
AD A061-222

* TM-115 Bryant, Randal E. and Jack B. Dennis
Concurrent Programming, October 1978.
AD A061-180

TM-116 Pratt, Vaughan R.
Applications of Modal Logic to Programming, December 1978.

TM-117 Pratt, Vaughan R.
Six Lectures on Dynamic Logic, December 1978.

* TM-118 Borkin, Sheldon A.
Data Model Equivalence, December 1978.
AD A062-753

TM-119 Shamir, Adi and Richard E. Zippel
On the Security of the Merkle-Hellman Cryptographic Scheme, December 1978.
AD A063-104

* TM-120 Brock, Jarvis D.
Operational Semantics of a Data Flow Language, S.M. Thesis, EE & CS Dept., December 1978.
AD A062-997

* TM-121 Jaffe, Jeffrey
The Equivalence of R. E. Programs and Data Flow Schemes, January 1979.

TM-122 Jaffe, Jeffrey
Efficient Scheduling of Tasks Without Full Use of Processor Resources, January 1979.

TM-123 Perry, Harold M.
An Improved Proof of the Rabin-Hartmanis-Stearns Conjecture, S.M. & E.E. Thesis, EE & CS Dept., January 1979.

TM-124 Toffoli, Tommaso
Bicontinuous Extensions of Invertible Combinatorial Functions, January 1979.
AD A063-886

TM-125 Shamir, Adi, Ronald L. Rivest and Leonard M. Adleman
Mental Poker, February 1979.
AD A066-331

TM-126 Meyer, Albert R., and Michael S. Paterson
With What Frequency Are Apparently Intractable Problems Difficult?, February 1979.

TM-127 Strazdas, Richard J.
A Network Traffic Generator for Decnet, S.B. & S.M. Thesis, EE & CS Dept., March 1979.

TM-128 Loui, Michael C.
Minimum Register Allocation is Complete in Polynomial Space, March 1979.

TM-129 Shamir, Adi
On the Cryptocomplexity of Knapsack Systems, April 1979.
AD A067-972

TM-130 Greif, Irene and Albert R. Meyer
Specifying the Semantics of While-Programs: A Tutorial and Critique of a Paper by Hoare and Lauer, April 1979.
AD A068-967

* TM-131 Adleman, Leonard M.
Time, Space and Randomness, April 1979.

TM-132 Patil, Ramesh S.
Design of a Program for Expert Diagnosis of Acid Base and Electrolyte Disturbances, May 1979.

TM-133 Loui, Michael C.
The Space Complexity of Two Pebble Games on Trees, May 1979.

TM-134 Shamir, Adi
How to Share a Secret, May 1979.
AD A069-397

TM-135 Wyleczuk, Rosanne H.
Timestamps and Capability-Based Protection in a Distributed Computer Facility, S.B. & S.M. Thesis, EE & CS Dept., June 1979.

TM-136 Misunas, David P.
Report on the Second Workshop on Data Flow Computer and Program Organization, June 1979 .

TM-137 Davis, Ernest and Jeffrey M. Jaffe
Algorithms for Scheduling Tasks on Unrelated Processors, June 1979.

TM-138 Pratt, Vaughan R.
Dynamic Algebras: Examples, Constructions, Applications, July 1979.

TM-139 Martin, William A.
Roles, Co-Descriptors, and the Formal Representation of Quantified English Expressions (Revised May 1980), September 1979.
AD A074-625

TM-140 Szolovits, Peter
Artificial Intelligence and Clinical Problem Solving, September 1979.

TM-141 Hammer, Michael and Dennis McLeod
On Database Management System Architecture, October 1979.
AD A076-417

* TM-142 Lipski, Witold, Jr.
On Data Bases with Incomplete Information, October 1979.

TM-143 Leth, James W.
An Intermediate Form for Data Flow Programs, S.M. Thesis, EE & CS Dept., November 1979.

TM-144 Takagi, Akihiro
Concurrent and Reliable Updates of Distributed Databases, November 1979.

TM-145 Loui, Michael C.
A Space Bound for One-Tape Multidimensional Turing Machines, November 1979.

TM-146 Aoki, Donald J.
A Machine Language Instruction Set for a Data Flow Processor, S.M. Thesis, EE & CS Dept., December 1979.

TM-147 Schroeppel, Richard and Adi Shamir
A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems, January 1980.
AD A080-385

TM-148 Adleman, Leonard M., and Michael C. Loui
Space-Bounded Simulation of Multitape Turing Machines, January 1980.

TM-149 Pallottino, Stefano and Tommaso Toffoli
An Efficient Algorithm for Determining the Length of the Longest Dead Path in an "Lifo" Branch-and-Bound Exploration Schema, January 1980.
AD A079-912

TM-150 Meyer, Albert R.
Ten Thousand and One Logics of Programming, February 1980.

TM-151 Toffoli, Tommaso
Reversible Computing, February 1980.
AD A082-021

TM-152 Papadimitriou, Christos H.
On the Complexity of Integer Programming, February 1980.

TM-153 Papadimitriou, Christos H.
Worst-Case and Probabilistic Analysis of a Geometric Location Problem, February 1980.

TM-154 Karp, Richard M., and Christos H. Papadimitriou
On Linear Characterizations of Combinatorial Optimization Problems, February 1980.

TM-155 Atai, Alon, Richard J. Lipton, Christos H. Papadimitriou and M. Rodeh,
Covering Graphs by Simple Circuits, February 1980.

TM-156 Meyer, Albert R., and Rohit Parikh
Definability in Dynamic Logic, February 1980.

TM-157 Meyer, Albert R., and Karl Winklmann
On the Expressive Power of Dynamic Logic, February 1980.

TM-158 Stark, Eugene W.
Semaphore Primitives and Starvation-Free Mutual Exclusion, S.M. Thesis, EE & CS Dept., March 1980.

TM-159 Pratt, Vaughan R.
Dynamic Algebras and the Nature of Induction, March 1980.

TM-160 Kanellakis, Paris C.
On the Computational Complexity of Cardinality Constraints in Relational Databases, March 1980.

TM-161 Lloyd, Errol L.
Critical Path Scheduling of Task Systems with Resource and Processor Constraints, March 1980.

TM-162 Marcum, Alan M.
A Manager for Named, Permanent Objects, S.B. & S.M. Thesis, EE & CS Dept., April 1980.
AD A083-491

TM-163 Meyer, Albert R., and Joseph Y. Halpern
Axiomatic Definitions of Programming Languages: A Theoretical Assessment, April 1980.

TM-164 Shamir, Adi
The Cryptographic Security of Compact Knapsacks (Preliminary Report), April 1980.
AD A084-456

TM-165 Finseth, Craig A.
Theory and Practice of Text Editors or A Cookbook for an Emacs, S.B. Thesis, EE & CS Dept., May 1980.

TM-166 Bryant, Randal E.
Report on the Workshop on Self-Timed Systems, May 1980.

TM-167 Pavelle, Richard and Michael Wester
Computer Programs for Research in Gravitation and Differential Geometry, June 1980.

Technical Reports

TR-143 Silverman, Howard
A Digitalis Therapy Advisor, S.M. Thesis, EE Dept., January 1975.

TR-144 Rackoff, Charles
The Computational Complexity of Some Logical Theories, Ph.D. Dissertation, EE Dept., February 1975.

* TR-145 Henderson, D. Austin
The Binding Model: A Semantic Base for Modular Programming Systems, Ph.D. Dissertation, EE Dept., February 1975.
AD A006-961

TR-146 Malhotra, Ashok
Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis, Ph.D. Dissertation, EE Dept., February 1975.

TR-147 Van De Vanter, Michael L.
A Formalization and Correctness Proof of the CGOL Language System, S.M. Thesis, EE Dept., March 1975.

TR-148 Johnson, Jerry
Program Restructuring for Virtual Memory Systems, Ph.D. Dissertation, EE Dept., March 1975.
AD A009-218

* TR-149 Snyder, Alan
A Portable Compiler for the Language C, S.B. & S.M. Thesis, EE Dept., May 1975.
AD A010-218

* TR-150 Rumbaugh, James E.
A Parallel Asynchronous Computer Architecture for Data Flow Programs, Ph.D. Dissertation, EE Dept., May 1975.
AD A010-918

* TR-151 Manning, Frank B.
Automatic Test, Configuration, and Repair of Cellular Arrays, Ph.D. Dissertation, EE Dept., June 1975.
AD A012-822

TR-152 Qualitz, Joseph E.
Equivalence Problems for Monadic Schemas, Ph.D. Dissertation, EE Dept., June 1975.
AD A012-823

TR-153 Miller, Peter B.
Strategy Selection in Medical Diagnosis, S.M. Thesis, EE & CS Dept., September 1975.

TR-154 Greif, Irene
Semantics of Communicating Parallel Processes, Ph.D. Dissertation, EE & CS Dept., September 1975.
AD A016-302

* TR-155 Kahn, Kenneth M.
Mechanization of Temporal Knowledge, S.M. Thesis, EE & CS Dept., September 1975.

TR-156 Bratt, Richard G.
Minimizing the Naming Facilities Requiring Protection in a Computer Utility, S.M. Thesis, EE & CS Dept., September 1975.

* TR-157 Meldman, Jeffrey A.
A Preliminary Study in Computer-Aided Legal Analysis, Ph.D. Dissertation, EE & CS Dept., November 1975.
AD A018-997

TR-158 Grossman, Richard W.
Some Data-base Applications of Constraint Expressions, S.M. Thesis, EE & CS Dept., February 1976.
AD A024-149

* TR-159 Hack, Michel
Petri Net Languages, March 1976.

TR-160 Bosyj, Michael
A Program for the Design of Procurement Systems, S.M. Thesis, EE & CS Dept., May 1976.
AD A026-688

* TR-161 Hack, Michel
Decidability Questions, Ph.D. Dissertation, EE & CS Dept., June 1976.

TR-162 Kent, Stephen T.
Encryption-Based Protection Protocols for Interactive User-Computer Communication, S.M. Thesis, EE & CS Dept., June 1976.
AD A026-911

* TR-163 Montgomery, Warren A.
A Secure and Flexible Model of Process Initiation for a Computer Utility, S.M. & E.E. Thesis, EE & CS Dept., June 1976.

TR-164 Reed, David P.
Processor Multiplexing in a Layered Operating System, S.M. Thesis, EE & CS Dept., July 1976.

TR-165 McLeod, Dennis J.
High Level Expression of Semantic Integrity Specifications in a Relational Data Base System, S.M. Thesis, EE & CS Dept., September 1976.
AD A034-184

TR-166 Chan, Arvola Y.
Index Selection in a Self-Adaptive Relational Data Base Management System, S.M. Thesis, EE & CS Dept., September 1976.
AD A034-185

* TR-167 Janson, Philippe A.
Using Type Extension to Organize Virtual Memory Mechanisms, Ph.D. Dissertation, EE & CS Dept., September 1976.

TR-168 Pratt, Vaughan R.
Semantical Considerations on Floyd-Hoare Logic, September 1976.

TR-169 Safran, Charles, James F. Desforges and Philip N. Tsichlis
Diagnostic Planning and Cancer Management, September 1976.

TR-170 Furtek, Frederick C.
The Logic of Systems, Ph.D. Dissertation, EE & CS Dept., December 1976.

TR-171 Huber, Andrew R.
A Multi-Process Design of a Paging System, S.M. & E.E. Thesis, EE & CS Dept., December 1976.

TR-172 Mark, William S.
The Reformulation Mo[illegible]ertise, Ph.D. Dissertation, EE & CS Dept., December 19
AD A035-397

TR-173 Goodman, Nathan
Coordination of Parallel Processes in the Actor Model of Computation, S.M. Thesis, EE & CS Dept., December 1976.

TR-174 Hunt, Douglas H.
A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem, S.M. & E.E. Thesis, EE & CS Dept., December 1976.

TR-175 Goldberg, Harold J.
A Robust Environment for Program Development, S.M. Thesis, EE & CS Dept., February 1977.

TR-176 Swartout, William R.
A Digitalis Therapy Advisor with Explanations, S.M. Thesis, EE & CS Dept., February 1977.

* TR-177 Mason, Andrew H.
A Layered Virtual Memory Manager, S.M. & E.E. Thesis, EE & CS Dept., May 1977.

* TR-178 Bishop, Peter B.
Computer Systems with a Very Large Address Space and Garbage Collection, Ph.D. Dissertation, EE & CS Dept., May 1977.
AD A040-601

TR-179 Karger, Paul A.
Non-Discretionary Access Control for Decentralized Computing Systems, S.M. Thesis, EE & CS Dept., May 1977.
AD A040-804

TR-180 Luniewski, Allen W.
A Simple and Flexible System Initialization Mechanism, S.M. & E.E. Thesis, EE & CS Dept., May 1977.

TR-181 Mayr, Ernst W.
The Complexity of the Finite Containment Problem for Petri Nets, S.M. Thesis, EE & CS Dept., June 1977 .

TR-182 Brown, Gretchen P.
A Framework for Processing Dialogue, June 1977.
AD A042-370

TR-183 Jaffe, Jeffrey M.
Semilinear Sets and Applications, S.M. Thesis, EE & CS Dept., July 1977.

* TR-184 Levine, Paul H.
Facilitating Interprocess Communication in a Heterogeneous Network Environment, S.B. & S.M. Thesis, EE & CS Dept., July 1977.
AD A043-901

TR-185 Goldman, Barry
Deadlock Detection in Computer Networks, S.B. & S.M. Thesis, EE & CS Dept., September 1977.
AD A047-025

TR-186 Ackerman, William B.
A Structure Memory for Data Flow Computers, S.M. Thesis, EE & CS Dept., September 1977.
AD A047-026

TR-187 Long, William J.
A Program Writer, Ph.D. Dissertation, EE & CS Dept., November 1977.
AD A047-595

TR-188 Bryant, Randal E.
Simulation of Packet Communication Architecture Computer Systems, S.M. Thesis, EE & CS Dept., November 1977.
AD A048-290

TR-189 Ellis, David J.
Formal Specifications for Packet Communication Systems, Ph.D. Dissertation, EE & CS Dept., November 1977.
AD A048-980

TR-190 Moss, J. Eliot B.
Abstract Data Types in Stack Based Languages, S.M. Thesis, EE & CS Dept., February 1978.
AD A052-332

TR-191 Yonezawa, Akinori
Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, Ph.D. Dissertation, EE & CS Dept., January 1978.
AD A051-149

TR-192 Niamir, Bahram
Attribute Partitioning in a Self-Adaptive Relational Database System, S.M. Thesis, EE & CS Dept., January 1978.
AD A053-292

TR-193 Schaffert, J. Craig
A Formal Definition of CLU, S.M. Thesis, EE & CS Dept., January 1978.

TR-194 Hewitt, Carl and Henry Baker, Jr.
Actors and Continuous Functionals, February 1978.
AD A052-266

TR-195 Bruss, Anna R.
On Time-Space Classes and Their Relation to the Theory of Real Addition, S.M. Thesis, EE & CS Dept., March 1978.

* TR-196 Schroeder, Michael D., David D. Clark, Jerome H. Saltzer and Douglas H. Wells
Final Report of the Multics Kernel Design Project, March 1978.

* TR-197 Baker, Henry Jr.
Actor Systems for Real-Time Computation, Ph.D. Dissertation, EE & CS Dept., March 1978.
AD A053-328

TR-198 Halstead, Robert H., Jr.
Multiple-Processor Implementation of Message-Passing Systems, S.M. Thesis, EE & CS Dept., April 1978.
AD A054-009

* TR-199 Terman, Christopher J.
The Specification of Code Generation Algorithms, S.M. Thesis, EE & CS Dept., April 1978.
AD A054-301

TR-200 Harel, David
Logics of Programs: Axiomatics and Descriptive Power, Ph.D. Dissertation, EE & CS Dept., May 1978.

TR-201 Scheifler, Robert W.
A Denotational Semantics of CLU, S.M. Thesis, EE & CS Dept., June 1978.

* TR-202 Principato, Robert N., Jr.
A Formalization of the State Machine Specification Technique, S.M. & E.E. Thesis, EE & CS Dept., July 1978.

TR-203 Laventhal, Mark S.
Synthesis of Synchronization Code for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1978.
AD A058-232

TR-204 Teixeira, Thomas J.
Real-Time Control Structures for Block Diagram Schemata, S.M. Thesis, EE & CS Dept., August 1978.
AD A061-122

TR-205 Reed, David P.
Naming and Synchronization in a Decentralized Computer System, Ph.D. Dissertation, EE & CS Dept., October 1978.
AD A061-407

TR-206 Borkin, Sheldon A.
Equivalence Properties of Semantic Data Models for Database Systems, Ph.D. Dissertation, EE & CS Dept., January 1979.
AD A066-386

* TR-207 Montgomery, Warren A.
Robust Concurrency Control for a Distributed Information System, Ph.D. Dissertation, EE & CS Dept., January 1979.
AD A066-996

TR-208 Krizan, Brock C.
A Minicomputer Network Simulation System, S.B. & S.M. Thesis, EE & CS Dept., February 1979.

TR-209 Snyder, Alan
A Machine Architecture to Support an Object-Oriented Language, Ph.D. Dissertation, EE & CS Dept., March 1979.
AD A068-111

TR-210 Papadimitriou, Christos H.
Serializability of Concurrent Database Updates, March 1979.

* TR-211 Bloom, Toby
Synchronization Mechanisms for Modular Programming Languages, S.M. Thesis, EE & CS Dept., April 1979.
AD A069-819

TR-212 Rabin, Michael O.
Digitalized Signatures and Public-Key Functions as Intractable as Factorization, March 1979.

TR-213 Rabin, Michael O.
Probabilistic Algorithms in Finite Fields, March 1979.

TR-214 McLeod, Dennis
A Semantic Data Base Model and Its Associated Structured User Interface, Ph.D. Dissertation, EE & CS Dept., March 1979.
AD A068-112

TR-215 Svobodova, Liba, Barbara Liskov and David Clark
Distributed Computer Systems: Structure and Semantics, April 1979.
AD A070-286

TR-216 Myers, John M.
Analysis of the SIMPLE Code for Dataflow Computation, June 1979.

TR-217 Brown, Donna J.
Storage and Access Costs for Implementations of Variable - Length Lists, Ph.D. Dissertation, EE & CS Dept., June 1979.

TR-218 Ackerman, William B. and Jack B. Dennis
VAL--A Value-Oriented Algorithmic Language Preliminary Reference Manual, June 1979.
AD A072-394

* TR-219 Sollins, Karen R.
Copying Complex Structures in a Distributed System, S.M. Thesis, EE & CS Dept., July 1979.
AD A072-441

* TR-220 Kosinski, Paul R.
Denotational Semantics of Determinate and Non-Determinate Data Flow Programs, Ph.D. Dissertation, EE & CS Dept., July 1979.

TR-221 Berzins, Valdis A.
Abstract Model Specifications for Data Abstractions, Ph.D. Dissertation, EE & CS Dept., July 1979.

* TR-222 Halstead, Robert H., Jr.
Reference Tree Networks: Virtual Machine and Implementation, Ph.D. Dissertation, EE & CS Dept., September 1979.
AD A076-570

TR-223 Brown, Gretchen P.
Toward a Computational Theory of Indirect Speech Acts, October 1979.
AD A077-065

TR-224 Isaman, David L.
Data-Structuring Operations in Concurrent Computations, Ph.D. Dissertation, EE & CS Dept., October 1979.

TR-225 Liskov, Barbara, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler and Alan Snyder
CLU Reference Manual, October 1979.
AD A077-018

TR-226 Reuveni, Asher
The Event Based Language and Its Multiple Processor Implementations, Ph.D. Dissertation, EE & CS Dept., January 1980.
AD A081-950

TR-227 Rosenberg, Ronni L.
Incomprehensible Computer Systems: Knowledge Without Wisdom, S.M. Thesis, EE & CS Dept., January 1980.

TR-228 Weng, Kung-Song
An Abstract Implementation for a Generalized Data Flow Language, Ph.D. Dissertation, EE & CS Dept., January 1980.

TR-229 Atkinson, Russell R.
Automatic Verification of Serializers, Ph.D. Dissertation, EE & CS Dept., March 1980.
AD A082-885

TR-230 Baratz, Alan E.
The Complexity of the Maximum Network Flow Problem, S.M. Thesis, EE & CS Dept., March 1980.

TR-231 Jaffe, Jeffrey M.
Parallel Computation: Synchronization, Scheduling, and Schemes, Ph.D. Dissertation, EE & CS Dept., March 1980.

TR-232 Luniewski, Allen W.
The Architecture of an Object Based Personal Computer, Ph.D. Dissertation, EE & CS Dept., March 1980.
AD A083-433

TR-233 Kaiser, Gail E.
Automatic Extension of an Augmented Transition Network Grammar for Morse Code Conversations, S.B. Thesis, EE & CS Dept., April 1980.
AD A084-411

TR-234 Herlihy, Maurice P.
Transmitting Abstract Values in Messages, S.M. Thesis, EE & CS Dept., May 1980.
AD A086-984

TR-235 Levin, Leonid A.
A Concept of Independence with Applications in Various Fields of Mathematics, May 1980.

TR-236 Lloyd, Errol L.
Scheduling Task Systems with Resources, Ph.D. Dissertation, EE & CS Dept., May 1980.

TR-237 Kapur, Deepak
Towards a Theory for Abstract Data Types, Ph.D. Dissertation, EE & CS Dept., June 1980.
AD A085-877

TR-238 Bloniarz, Peter A.
The Complexity of Monotone Boolean Functions and an Algorithm for Finding Shortest Paths in a Graph, Ph.D. Dissertation, EE & CS Dept., June 1980.

TR-239 Baker, Clark M.
Artwork Analysis Tools for VLSI Circuits, S.M. & E.E. Thesis, EE & CS Dept., June 1980.
AD A087-040

Progress Reports

* Project MAC Progress Report I, to July 1964
  AD 465-088

* Project MAC Progress Report II, July 1964-July 1965
  AD 629-494

* Project MAC Progress Report III, July 1965-July 1966
  AD 648-346

* Project MAC Progress Report IV, July 1966-July 1967
  AD 681-342

* Project MAC Progress Report V, July 1967-July 1968
  AD 687-770

* Project MAC Progress Report VI, July 1968-July 1969
  AD 705-434

* Project MAC Progress Report VII, July 1969-July 1970
  AD 732-767

* Project MAC Progress Report VIII, July 1970-July 1971
  AD 735-148

* Project MAC Progress Report IX, July 1971-July 1972
  AD 756-689

* Project MAC Progress Report X, July 1972-July 1973
  AD 771-428

* Project MAC Progress Report XI, July 1973-July 1974
AD A004-966

* Laboratory for Computer Science Progress Report XII, July 1974-July 1975
AD A024-527

* Laboratory for Computer Science Progress Report XIII, July 1975-July 1976
AD A061-246

Laboratory for Computer Science Progress Report XIV, July 1976-July 1977
AD A061-932

Laboratory for Computer Science Progress Report 15, July 1977-July 1978
AD A073-958

Laboratory for Computer Science Progress Report 16, July 1978-July 1979
AD A088-355

Copies of all reports with AD and PB numbers listed in Publications may be secured from the National Technical Information Service, Operations Division, Springfield, Virginia 22151. Prices vary. The AD or PB number must be supplied with the request.

* Out of Print reports may be obtained from NTIS if the AD number is supplied (see above). Out of Print reports without an AD or PB number are unobtainable.

OFFICIAL DISTRIBUTION LIST

Director
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

Attention: Program Management

2 copies

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention: Marvin Denicoff, Code 437

3 copies

Office of Naval Research
Resident Representative
Massachusetts Institute of Technology
Building E19-628
Cambridge, Mass. 02139

Attention: A. Forrester

1 copy

Director
Naval Research Laboratory
Washington, D.C. 20375

Attention: Code 2627

6 copies

Defense Documentation Center
Bldg. 5, Cameron Station
Alexandria, Virginia 22213

12 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, Mass. 02210

1 copy

END